Molekulare Algorithmen „DNA Computing“

Thema 14

Programmierung eines Simulators für die Arbeitsschritte und die erzeugte Sprache einer frei vom Nutzer konfigurierbaren Chomsky-Grammatik

Fleming Kretschmer | Mark Kriegbaum

Aufgabe

- Programmieren Sie in einer Programmiersprache Ihrer Wahl (Java, Python, . . .) einen Simulator für beliebig konfigurierbare Chomsky-Grammatiken. Nach jedem Ableitungsschritt sollen die entstandenen Zeichenfolgen angezeigt werden sowie zusätzlich alle bis dahin bereits generierten Wörter der erzeugten Sprache.
- Die Konfiguration der Chomsky-Grammatik kann durch Einlesen einer Textdatei erfolgen, in welcher die Menge der Variablensymbole, die Menge der Terminalsymbole, die Ersetzungsregeln sowie das Startsymbol vorgegeben werden.
- Eine einfache Textausgabe am Monitor sowie zusätzlich in eine Textdatei ist ausreichend. Der Nutzer gibt zudem vor, über wieviele Zeitschritte die Grammatik simuliert wird.
- Demonstrieren Sie Ihr Programm anhand mehrerer Fallstudien.

chomsky.py

- Als Python Skript implementiert
- Attribute der Klasse chomsky: var_symbols, term_symbols, rules, start_symbols

```python
def __init__(self, var_symbols, term_symbols, rules, start_symbol,
             verbose=False, progress=False):
    self.var_symbols = var_symbols
    self.term_symbols = term_symbols
    if (len(set(var_symbols).intersection(set(term_symbols))) > 0):
        raise Exception('Nonterminal symbols and terminal symbols '
                        'cannot contain the same symbols!')
    self.rules = rules      # = [('A', 'aa'), ('S', 'Aa')]
    self.start_symbol = start_symbol
    self.verbose = verbose
    self.progress = progress
```

simulate()

```python
def simulate(self, steps=10, word_iter_thr=0, nr_words=0):
    """Starting with the start_symbol, applies randomly chosen rules in
    every step and returns all produced words (not containing nonterminal
    symbols)
    """
    words = set()
    word = self.start_symbol
    word_rules = {}          # rules applied for each word
    applied_rules = []
    rules_random = self.rules.copy()
    word_iter_counter = 0
    stuck = False
    if (word_iter_thr == 0):
        word_iter_thr = steps
    if (steps == 0):
        from itertools import count
        steps_iter = count()
    else:
        steps_iter = range(steps)
    if (self.progress):
        from tqdm import tqdm
        steps_iter = tqdm(steps_iter)
```

simulate()

```python
for step in steps_iter:
    word_iter_counter += 1
    if (word_iter_thr != 0 and word_iter_counter > word_iter_thr):
        word = self.start_symbol
        word_iter_counter = 0
    shuffle(rules_random)
    for i, (rule_l, rule_r) in enumerate(rules_random):
        if (rule_l in word):
            applied_rules.append((rule_l, rule_r))
            word_before = word
            word = word.replace(rule_l, rule_r, 1)
            # output
            if (self.verbose):
                print('[{}] {}: {} -> {}'.format(step+1,
                                                 (rule_l, rule_r),
                                                 word_before, word))
            break
    else:
        # All rules have been tried, none could be applied
        stuck = True
```

simulate()

```python
        else:
            # All rules have been tried, none could be applied
            stuck = True
        if (not any(var_symbol in word for var_symbol in self.var_symbols)):
            # word only contains terminal chars -> is a valid word
            words.add(word)
            word_rules[word] = applied_rules
            applied_rules = []
            word = self.start_symbol
        if (stuck):
            word = self.start_symbol
            stuck = False
        if (self.verbose):
            print('Words generated thus far ({}): '.format(len(words))
                  + ', '.join("'"+word+"'" for word in sorted(words)))
        if (nr_words != 0 and len(words) >= nr_words):
            break
    return(words, word_rules)
```

weitere Methoden

```python
def replace_rules_(old_rules, replace_rules):
    new_rules = []
    for rule_l, rule_r in old_rules:
        for before, after in replace_rules:
            if (before in rule_l or before in rule_r):
                rule_l = rule_l.replace(before, after)
                rule_r = rule_r.replace(before, after)
        new_rules.append((rule_l, rule_r))
    return new_rules


def read_dict(filename):
    with open(filename) as handle:
        return(json.load(handle))


def write_file(words, word_rules, filename):
    """Writes generated words and rules applied for each
    to output file
    """
    with open(filename, 'w') as handle:
        for word in words:
            handle.write(word + ':\t'
                         + ', '.join(str(rule) for rule in word_rules[word])
                         + '\n')
```

CLI

```
python chomsky.py -h
usage: chomsky.py [-h] [-s N] [-w #w] [--word_thr T]
                  [-r find repl [find repl ...]] [-o f] [-V] [-p]
                  [-v VALIDATE] [--val_setup VAL_SETUP]
                  grammar

Simulation of Chomsky grammars

positional arguments:
  grammar               Chomsky grammar as JSON file

optional arguments:
  -h, --help            show this help message and exit
  -s N, --steps N       number of iterations to perform (0: ∞) (default: 100)
  -w #w, --words #w     number of words to generate, when the number is
                        reached, the program terminates (0: ∞) (default: 0)
  --word_thr T          Maximum number of rule applications for one word (0:
                        ∞) (default: 0)
  -r find repl [find repl ...], --replace_rules find repl [find repl ...]
                        Allows replacement of variable symbols in rules
                        (default: None)
  -o f, --output_file f
                        allows writing to specified output file (default:
                        None)
  -V, --verbose         verbose output (default: False)
  -p, --progress        displays progressbar in simulation {Requires tqdm
                        package} (default: False)
  -v VALIDATE, --validate VALIDATE
                        lambda function to validate words (default: None)
  --val_setup VAL_SETUP
                        extra code(e.g., imports needed for the validation)
                        (default: None)
```

Input

- Als JSON Datei

```
{"var_symbols": ["S", "A", "B"],
 "term_symbols": ["a", "b"],
 "rules": [["S", "AB"], ["A", "aAA"], ["A", ""],
           ["B", "bBB"], ["B", ""]],
 "start_symbol": "S"}
```

Beispiel 1 – Logarithmus

```
python chomsky.py -s 0 -w 1 grammar_log.json -r 1 NNNNNNNNNNNNNNNNNNNNNNNNNNN 2 BBB
```

```
{"start_symbol": "S",
 "var_symbols": ["L", "1", "I", "2", "R", "D", "N", "B", "X", "M", "Z"],
 "term_symbols": ["a"],
 "rules": [["S", "L1I2RD"],
           ["LNIBB", "X"],
           ["XB", "X"],
           ["XRD", ""],
           ["NNIBB", "NNBB"],
           ["NB", "M"],
           ["NMB", "M"],
           ["LMB", "LM"],
           ["NMR", "N2RM"],
           ["LMR", "LRZM"],
           ["RZM", "NRZ"],
           ["NNRZD", "NN2RDZ"],
           ["LNRZD", "a"],
           ["aZ", "aa"]]
}
```

Test für die Logarithmusberechnung

- Grammatik:

```
{"start_symbol": "S",
 "var_symbols": ["L", "1", "I", "2", "R", "D", "N", "B", "X", "M", "Z"],
 "term_symbols": ["a"],
 "rules": [["S", "L1I2RD"],
           ["LNIBB", "X"],
           ["XB", "X"],
           ["XRD", ""],
           ["NNIBB", "NNBB"],
           ["NB", "M"],
           ["NMB", "M"],
           ["LMB", "LM"],
           ["NMR", "N2RM"],
           ["LMR", "LRZM"],
           ["RZM", "NRZ"],
           ["NNRZD", "NN2RDZ"],
           ["LNRZD", "a"],
           ["aZ", "aa"]]
}
```

- unittest:

```
def test_log_grammar(self):
    raw_dict = chomsky.read_dict('grammar_log.json')
    steps = np.full((10**3, 6), np.nan)
    for base in range(4,6):
        for number in tqdm(range(0, 10**3, base)):
            chomsky_dict = raw_dict.copy()
            chomsky_dict['rules'] = chomsky.replace_rules_(
                chomsky_dict['rules'],
                [('1', 'N'*number), ('2', 'B'*base)])
            c = chomsky.Chomsky(**chomsky_dict)
            words, applied_rules = c.simulate(steps=10**4, nr_words=1)
            if (len(words) == 0):
                continue
            word = words.pop()
            applied_rules = applied_rules[word]
            self.assertEqual(len(word), np.ceil(
                round(log(number, base), 4)),  # log is not that accurate
                            'number: {}, base: {}'.format(number, base))
            steps[number, base] = len(applied_rules)
    steps_df = pd.DataFrame(steps)
    steps_df.to_csv('log_steps.txt', na_rep='-')
```

Beispiel 2 – 3er Potenz

```
python chomsky.py -s 500000 -p grammar_cubic.json
--val_setup "from numpy import cbrt" -v "cbrt(len(x.strip('$')))"
```

```json
{
  "start_symbol": "S",
  "var_symbols": ["S", "U", "A", "B", "R", "L", "F"],
  "term_symbols": ["x", "$"],
  "rules": [
    ["S", "$U$"],
    ["S", "$$"],
    ["U", "ARF"],
    ["U", "AURF"],
    ["FR","RF"],
    ["F$","L$"],
    ["FL","LF"],
    ["RL","LRB"],
    ["RA","AR"],
    ["RB","BR"],
    ["R$","$"],
    ["AL","A"],
    ["BL","LB"],
    ["AB","BAx"],
    ["Ax","xA"],
    ["xB","Bx"],
    ["A$","$"],
    ["$B","$"]
  ]
}
```

Test für die 3er Potenz-Erzeugung

- Grammatik:

```
{
  "start_symbol": "S",
  "var_symbols": ["S", "U", "A", "B", "R", "L", "F"],
  "term_symbols": ["x", "$"],
  "rules": [
    ["S", "$U$"],
    ["S", "$$"],
    ["U", "ARF"],
    ["U", "AURF"],
    ["FR","RF"],
    ["F$","L$"],
    ["FL","LF"],
    ["RL","LRB"],
    ["RA","AR"],
    ["RB","BR"],
    ["R$","$"],
    ["AL","A"],
    ["BL","LB"],
    ["AB","BAx"],
    ["Ax","xA"],
    ["xB","Bx"],
    ["A$","$"],
    ["$B","$"]
  ]
}
```

- unittest:

```
def test_cubic_grammar(self):
    for c in [chomsky.Chomsky(**cbrt_dict)
              for cbrt_dict in self.cbrt_dicts]:
        words, word_rules = c.simulate(steps=0, nr_words=5)
        failed_words = set()
        for word in words:
            number = len(word.strip('$'))
            cubic_root = np.cbrt(number)
            # cbrt is not that accurate
            rest = round(cubic_root - np.floor(cubic_root), 4)
            self.assertEqual(rest, 0)
            if (rest != 0):
                failed_words.add(number)
        self.assertEqual(len(failed_words), 0, str(failed_words))
        print(failed_words)
```

zukünftige Weiterentwicklungen

- GUI
- Parallelisieren
- Alternativer Simulations-algorithmus