BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF INFORMATION TECHNOLOGY
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF INFORMATION SYSTEMS
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

WATSON-CRICK MODELS FOR FORMAL LANGUAGE PROCESSING

WATSON-CRICK MODELY PRO ZPRACOVÁNÍ FORMÁLNÍCH JAZYKŮ

MASTER'S THESIS
DIPLOMOVÁ PRÁCE

AUTHOR
AUTOR PRÁCE
Bc. JAN HAMMER

SUPERVISOR
VEDOUCÍ PRÁCE
Ing. ZBYNĚK KŘIVKA, Ph.D.

BRNO 2022

Brno University of Technology
Faculty of Information Technology

Department of Information Systems (DIFS) Academic year 2021/2022

Master's Thesis Specification

20427

Student:	**Hammer Jan, Bc.**
Programme:	Information Technology
Field of study:	Information Systems
Title:	**Watson-Crick Models for Formal Language Processing**
Category:	Algorithms and Data Structures

Assignment:

1. Study unconventional models working with double-stranded strings with a focus on Watson-Crick automata and grammars.
2. According to the supervisor's instructions, study and design algorithms to answer the membership of a given sentence in the language defined by the given Watson-Crick model (grammar/automaton).
3. Implement the proposed algorithms as an application with an emphasis on time efficiency.
4. Test the application and experimentally evaluate it on at least 20 examples consulted with the supervisor. Discuss the possibility of parallelizing these algorithms.

Recommended literature:

- N. L. M. Zulkufli et al. The Computational Power of Watson-Crick Grammars: Revisited. In: International Conference on Bio-Inspired Computing: Theories and Applications, p. 215-225, 2016.
- N. L. M. Zulkufli et al. Generative Power and Closure Properties of Watson-Crick Grammars. Applied Computational Intelligence and Soft Computing, 12 p., 2016. DOI 10.1155/2016/9481971.
- N. L. M. Zulkufli et al. Watson-Crick Context-Free Grammars: Grammar Simplifications and a Parsing Algorithm. The Computer Journal 61(9), p. 1361-1373, 2018.

Detailed formal requirements can be found at https://www.fit.vut.cz/study/theses/

Supervisor:	**Křivka Zbyněk, Ing., Ph.D.**
Head of Department:	Kolář Dušan, doc. Dr. Ing.
Beginning of work:	November 1, 2021
Submission deadline:	May 18, 2022
Approval date:	October 26, 2021

Abstract

This work focuses on Watson-Crick languages inspired by DNA computing, their models and algorithms of deciding the language membership. It analyzes a recently introduced algorithm called WK-CYK and introduces a state space search algorithm which is based on regular Breath-first search but uses a number of optimizations and heuristics to be efficient in practical use and able to analyze inputs of greater lengths. The key parts are the heuristics for pruning the state space (detecting dead ends) and heuristics for choosing the most promising branches to continue the search.

These two algorithms have been tested with 20 different Watson-Crick grammars (40 including their Chomsky normal form versions). While WK-CYK is able to decide the language membership in a reasonable time for inputs of length of roughly 30–50 symbols and its performance is very consistent for all kinds of grammars and inputs, the state space search is usually (89–98 % of cases) more efficient and able to do the computation for inputs with lengths of hundreds or even thousands of symbols. Thus, the state space search has a potential to be a good tool for practical Watson-Crick membership testing and is a good basis to further build on and further improve the efficiency of the algorithm.

Abstrakt

Tato práce se zabývá Watson-Crickovými jazyky, které jsou inspirovány výpočty nad DNA, dále jejich modely a algoritmy pro rozhodování členství řetězců v těchto jazycích. Analyzuje nedávno představený algoritmus nazvaný WK-CYK a prezentuje algoritmus založený na prohledávání stavového prostoru, jehož základem je standardní prohledávání prostoru do šířky, ale používá množství optimalizací a heuristik, aby byl v praxi efektivnější a dokázal analyzovat delší vstupy. Klíčové jsou heuristiky pro prořezávání stavového prostoru (detekují slepé větve) a heuristiky pro výběr nejslibnějších větví pro další výpočet.

Tyto dva algoritmy jsou testovány na 20 různých Watson-Crickových gramatikách (40 včetně jejich verzí v Chomského normální formě). Zatímco WK-CYK je schopen rozhodnout členství v jazyce v rozumném čase u vstupů o délce zhruba 30–50 symbolů, jeho efektivnost je velmi konzistentní u různých gramatik a různých vstupů, algoritmus prohledávající stavový prostor je obvykle (v 89–98 % případů) efektivnější a je schopen provést výpočet pro vstupy s délkou o stovkách často i tisících symbolů. Tedy tento algoritmus má potenciál být vhodným nástrojem pro praktické použití při rozhodování členství ve Watson-Crickových jazycích a nabízí vhodný základ pro další vývoj a vylepšení, která by dále zvyšovala efektivitu algoritmu.

Keywords

Watson-Crick languages, formal grammars, DNA computing, state space search, language membership problem

Klíčová slova

Watson-Crickovy jazyky, formální gramatiky, DNA výpočty, prohledávání stavového prostoru, problém členství v jazyce

Reference

HAMMER, Jan. *Watson-Crick Models for Formal Language Processing.* Brno, 2022. Master's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor Ing. Zbyněk Křivka, Ph.D.

Rozšířený abstrakt

Tato práce se zabývá Watson-Crickovými jazyky, které jsou inspirovány výpočty nad DNA, dále jejich modely a především algoritmy pro rozhodování členství řetězců v těchto jazycích. Tyto jazyky pracují s dvojitými vstupními řetězci, jejichž symboly jsou spojeny komplementární relací.

Jeden z hlavních modelů pro tyto jazyky jsou Watson-Crickovy bezkontextové gramatiky. Nedávno představený algoritmus WK-CYK, který je modifikací algoritmu CYK pro bezkontextové gramatiky, je určen právě pro rozhodování členství v těchto jazycích. Tento algoritmus pracuje s gramatikami ve Watson-Crickově Chomského normální formě (WK-CNF), což je modifikace Chomského normální formy bezkontextových gramatik. V této práci je algoritmus analyzován, implementován, je ověřena jeho funkčnost a deklarovaná složitost $\mathcal{O}(n^6)$ vzhledem k délce vstupu.

Hlavním přínosem práce je použití algoritmu prohledávání stavového prostoru pro rozhodování členství ve Watson-Crickových jazycích. Jeho základem je standardní algoritmus BFS (prohledávání stavového prostoru do šířky), který byl rozšířen o několik optimalizací a heuristik, aby jeho efektivita byla zajímavá pro praktické použití. Kořen stavového stromu je počáteční symbol gramatiky a následníci jsou získáni aplikací všech možných pravidel gramatiky na první neterminální symbol ve slově.

Nejdůležitějším rozšířením oproti původnímu BFS je pět heuristik pro prořezávání stavového stromu. Ty každý uzel analyzují a rozhodují, jestli je možné, že daný uzel vede k hledanému řešení. Heuristiky kontrolují, zda není již ve slově příliš mnoho symbolů, přičemž počet terminálních symbolů se snížit nemůže a u neterminálních symbolů je počítán minimální počet symbolů, které z nich mohou vzejít. Dále je kontrolováno dodržení komplementární relace a správnost pořadí již vygenerovaných terminálních symbolů. Dále je zde použita sada heuristik pro výběr nejslibnějšího uzlu, pro další prohledávání. Obecně preferují slova s méně neterminálními symboly a slova, jejíchž terminální symboly se více shodují s vstupním řetězcem.

Pro testování bylo vybráno 20 různých Watson-Crickových gramatik, které byly použity jednak v původní formě a také po transformaci do WK-CNF. Testování bylo provedeno v několika fázích. Nejprve byly porovnány na všech gramatikách všechny dostupné heuristiky pro výběr uzlu a vybrána ta, která dosáhla nejlepšího celkového času, protože je možné mít aktivní vždy jen jednu z nich. Dále byly na všech gramatikách testovány postupně všechny heuristiky pro prořezávání stromu a porovnáváno, zda má algoritmus lepší výkon s danou heuristikou aktivní nebo vypnutou. Tímto způsobem byla získána konfigurace, která má nejlepší celkový výkon. S touto konfigurací proběhly opět na všech gramatikách testy, které zjišťovaly maximální délku vstupu, pro kterou algoritmus prohledávání stavového prostoru získá výsledek v daném časovém limitu (10 sekund). Podobné testování následně proběhlo i pro algoritmus WK-CYK.

Při testování se ukázalo, že WK-CYK je schopen v rozumném časovém horizontu rozhodnout členství v jazyce pro řetězce o délce 30–50 znaků. Jeho výhodou je konzistentnost — ve všech testovaných případech, pro různé gramatiky i vstupy byla časová složitost velice podobná.

Výhodou algoritmu prohledávání stavového prostoru je předně univerzálnost. Je schopen pracovat s Watson-Crickovými gramatikami v jakékoli formě a s libovolnou komplementární relací. Díky tomu není nutné gramatiky převádět do WK-CNF, což může vyústit v daleko komplikovanější gramatiku s více pravidly. Při testování gramatik v základní formě byl výpočet v 38 ze 40 testovacích případů (tedy 97.5 %) výrazně efektivnější — je takto možné rozhodnout členství řetězců s řádově stovkami, často i tisíci symbolů. Po zahrnutí

gramatik po transformaci do WK-CNF byl algoritmus efektivnější v 71 z 80 (tedy 88.75 %) testovacích případů. Další výhodou tohoto algoritmu je jeho konfigurovatelnost, je možné pro konkrétní gramatiku porovnat účinnost heuristik pro výběr uzlu a používat tu, která je pro daný případ nejvhodnější. Stejně tak u heuristik pro prořezávání stavového stromu může být vhodné pro danou gramatiku některé vypnout, čímž se efektivita dále zvýší.

Watson-Crick Models for Formal Language Processing

Declaration

I hereby declare that this Master's thesis was prepared as an original work by the author under the supervision of Mr. Křivka I have listed all the literary sources, publications and other sources, which were used during the preparation of this thesis.

. . . .
Jan Hammer
May 17, 2022

Contents

Chapter 1

Introduction

The ability to read DNA, to understand it or even to modify it, is certainly one of the ways that many people think will define the future. But in order to work with DNA, there needs to be a mathematical model that can actually do calculations with such structures and that is prepared to be run on computers. Moreover, working with this model must be efficient enough because genetic code has a huge number of digits.

This works follows the work of M. Zulkufli et al. [10], [12], [11] who have studied models for working with Watson-Crick languages and introduced the WK-CYK algorithm, a modification of the CYK algorithm, which works with Watson-Crick context-free grammars and is able to decide the membership problem for these languages. The stated complexity of this algorithm is $\mathcal{O}(n^6)$ with respect to the input length. However, with this complexity the algorithm still does not seem to be useful for practical DNA computations considering how long DNA code is.

Therefore this work introduces the state space search algorithm. While its theoretical complexity is not as good as in case of WK-CYK, it takes a more practical approach. In practice, thanks to various heuristics, it is very often able to decide the membership in languages defined by Watson-Crick context-free grammars of inputs far longer then what WK-CYK can handle.

Chapter 2 contains an overview of most common models for working with Watson-Crick languages. Chapter 3 discusses ways of deciding membership problem of those languages with the focus on the WK-CYK algorithm. Chapter 4 introduces the state space search algorithm and the heuristics and optimizations that make it more efficient. Chapter 5 focuses on the implementation of the state space search and is probably going to be useful to someone who wants to delve into the code and use it or further build on it. Finally, chapter 6 contains twenty grammars that were used for testing both state space search and WK-CYK algorithms in practice and presents results of these tests.

A integral component of this thesis is an implementation in the Python programming language of the state space search algorithm, the WK-CYK algorithm and a number of tests used to analyze the state and space complexities and to compare the algorithms.

Chapter 2

Models of Watson-Crick languages

A number of models working with double stranded sequences has been proposed. The purpose of this chapter is to present a motivation for using them and to summarize these models and some of their key attributes that will be important in later chapters.

2.1 DNA as an inspiration for Watson-Crick languages

The study of Watson-Crick models is motivated by DNA (deoxyribonucleic acid) computing. In order to study the DNA mathematically, i.e. to perform mathematical operations over it, it is necessary to work with a suitable abstraction — a model which captures its key characteristics. Specifically, there are two characteristics that the Watson-Crick models capture — the fact that the DNA is a double stranded chain and the Watson-Crick relation between DNA nucleotides.

The two fundamental models that are used to define a language in computer theory are grammars and automata. Several versions of both have been proposed but all of them work with these two characteristics in a very similar manner.

DNA consists of two chains of nucleotides, one of which is marked as $5'$ end and the other $3'$ end. The chains are connected by covalent bonds and together form a double helix (figure 2.1). These two chains are represented in the Watson-Crick automata by two reading heads which read two inputs independently but are controlled by the same states. Similarly, Watson-Crick grammars produce by their rules not just a chain of symbols, but two chains.

Each nucleotide contains one of the four nucleobases - cytosine (C), guanine (G), adenine (A) and thymine (T). These bases are always connected with their counterpart: cytosine with guanine and adenine with thymine. That means that whenever one of the four appears in a chain, its counterpart appears in the other chain in the corresponding place being bound together by the covalent bond. The Watson-Crick models therefore introduce a complementarity relation — a relation between symbols which must be kept in the whole input for it to be valid. Typically, this relation is symmetric ($aRb \Leftrightarrow bRa$) and covers the whole alphabet (every symbol must have at least one counterpart). Often every symbol has exactly one counterpart, just like in case of DNA. The relation is usually defined as an identity (i.e. each symbol is related to itself and only to itself) which is still somewhat similar to the DNA pairing.

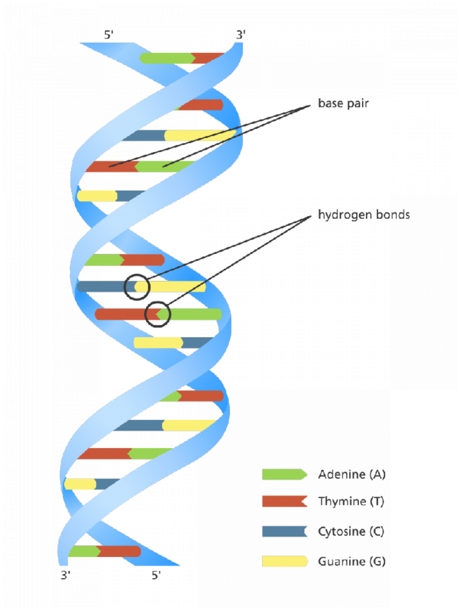

Figure 2.1: The DNA double helix

2.2 Watson-Crick automata

Watson-Crick automata have first been proposed in [7] as an enhancement of standard Finite Automata. Watson-Crick finite automaton is a 6-tuple $M = (V, \rho, Q, q_0, F, P)$ with the following meaning.

- V – finite input alphabet
- $\rho \subseteq V \times V$ – complementarity relation
- Q – finite set of states
- $q_0 \in Q$ – starting symbol
- $F \subseteq Q$ – set of final states
- P – finite set of transition rules in a form $q\binom{w_1}{w_2} \to q'$ where $q, q' \in Q, w_1, w_2 \in V^*$

Compared to Finite automata, Watson-Crick automata have different form of transition rules which read two strings at the same time. These represent the two independent reading heads — one reading the upper strand (w_1) and the other reading the lower strand (w_2). They also add the complementarity relation which is usually required to be symmetric. The symbols in the upper and lower strands with the same indexes need to adhere to it.

$\binom{w_1}{w_2}$ denotes simply a pair (w_1, w_2). A Watson-Crick domain is a set $WK_\rho(V)$ which denotes all valid double strands associated with a given V and ρ. Formally:

$$WK_\rho(V) = \begin{bmatrix} V \\ V \end{bmatrix}_\rho^* \quad \text{where} \quad \begin{bmatrix} V \\ V \end{bmatrix}_\rho = \left\{ \begin{bmatrix} a \\ b \end{bmatrix} | a, b \in V, (a, b) \in \rho \right\}$$

This implies that the both strands in a WK domain must have the same length.

A configuration of a Watson-Crick automaton is a pair $(q, \binom{w_1}{w_2})$ where $q \in Q$ is a current state and $w_1, w_2 \in V^*$ are the parts of the upper and lower strands yet to be read.

If $q\binom{u_1}{u_2} \to q' \in P$ and $\binom{u_1v_1}{u_2v_2} \in \binom{V^*}{V^*}$ then $q\binom{u_1v_1}{u_2v_2} \Rightarrow q'\binom{v_1}{v_2}$ is a transition of the Watson-Crick automaton. $\Rightarrow^*$ denotes the transitive and reflexive closure of the relation $\Rightarrow$.

A Watson-Crick automaton accepts the language $L(M)$:

$$L(M) = \left\{ w_1 \in V^* | q_0 \left[\begin{smallmatrix} w_1 \\ w_2 \end{smallmatrix}\right] \Rightarrow^* f\binom{\lambda}{\lambda} \text{ where } f \in F, w_2 \in V^*, \left[\begin{smallmatrix} w_1 \\ w_2 \end{smallmatrix}\right] \in WK_\rho(V) \right\}$$

where λ denotes a string of zero length (an empty string). This means that only the upper strand is accepted by this automaton to the language L. The lower strand has just an auxiliary purpose.

2.3 Special versions of Watson-Crick automata

Four special versions of Watson-Crick automata (WKA) are often used ([5], [13]). These are:

- stateless WKA — the WKA has only one state: $Q = F = q_0$
- all final WKA — all the states are final: $Q = F$
- simple WKA — each rule reads only one head: $(q\binom{w_1}{w_2} \to q' \in P) \Rightarrow (w_1 = \lambda \vee w_2 = \lambda)$
- 1-limited WKA — reads only one symbol at a time: $(q\binom{w_1}{w_2} \to q' \in P) \Rightarrow |w_1w_2| = 1$

Three of these four special types of WKAs have the same expressing power as the actual WKA, namely all final WKA, simple WKA and 1-limited WKA (stateless WKA is weaker). Therefore, one possible approach to decide membership would be to limit the decision algorithm to one of these three types without any loss in expressing power.

There are three different variants of deterministic WKA proposed in [5]. These are:

- Weakly deterministic WKA: WKA where in each reachable configuration, there is at most one possible continuation.
- Deterministic WKA: For any two rules which lead from the same state, either their upper strands or their lower strands must not be prefix comparable (one is not the prefix of the other). Formally: $(q\binom{u}{v} \to q_1 \in P \wedge q\binom{u'}{v'} \to q_2 \in P) \Rightarrow u \nsim_p u' \vee v \nsim_p v'$ where $\sim_p$ is the relation of prefix comparability.
- Strongly deterministic WKA: A deterministic WKA whose complementarity relation is identity.

It is not specified how to actually achieve weak determinism. In fact, [5] shows that this property is undecidable. Intuitively, for a WKA to be weakly deterministic but not deterministic, there must be at least two rules which could both be used in a certain configuration (otherwise it would be deterministic). But such a configuration must not be reachable (otherwise it would not be weakly deterministic). The configuration may be unreachable trivially — by rules using an unreachable state or a symbols that have no related symbols in the complementarity relation. But a configuration may be unreachable non-trivially, if it is possible to tell how many symbols will be read from each strand before reaching certain state.

Both weakly deterministic and deterministic WKA are in reality not deterministic (in an intuitive sense). Their determinism relies on the fact that the configuration is known and for the configuration to be known, the entire input (meaning both strands of the input) needs to be specified. But that is often not a way how WKA are used, since WKA decides the membership in a language for the upper strand only. That means that a compatible strand has to be found in the process of running the WKA. Theoretically, it is possible to approach this problem by first generating all possible lower strands for the given upper strand based solely on the complementarity relation and afterwards use all these pairs as inputs for the WKA. In such a case, the weakly deterministic and deterministic automata would be truly deterministic, however this is clearly not feasible for non-trivial complementarity relations. Therefore, the strongly deterministic WKA is the only one witch is truly deterministic under all circumstances because the identity relation requirement leaves no space for these types of non-determinism.

2.4 Watson-Crick grammars

The first kind of Watson-Crick grammars introduced were the Watson-Crick regular grammars [14]. The key features are shared with Watson-Crick automata. Specifically, it it the complementarity relation ρ and the double stranded strings that the grammar produces. The WK regular grammars have been used as a basis for Watson-Crick linear grammars and Watson-Crick context-free grammars introduced in [10]. Since a WK linear grammar is a generalization of WK regular grammar and WK context-free grammar is a generalization of WK linear grammar, it makes sense to start with the definition of the context-free version and then specify the constraints of linear and regular versions.

A **Watson-Crick context-free grammar** is $G = (N, T, \rho, P, S)$ where N is a finite set of non-terminals, T is a finite set of terminals and $N \cup T = \emptyset$, $S \in N$ is a starting non-terminal, $\rho \subseteq T \times T$ is a symmetric complementarity relation, and P is a finite set of rules that have the form $A \to \alpha$ where $A \in N \wedge \alpha \in (N \cup \binom{T^*}{T^*})^*$.

The derivation of the grammar G starts with the starting symbol S. $x \in (N \cup \binom{T^*}{T^*})^*$ directly derives $y \in (N \cup \binom{T^*}{T^*})^*$, denoted by $x \Rightarrow y$, if and only if:

$$x = \beta A \gamma \ \wedge \ y = \beta \alpha \gamma$$

where $A \in N \ \wedge \ \alpha, \beta, \gamma \in (N \cup \binom{T^*}{T^*})^* \ \wedge \ A \to \alpha \in P$.

The language generated by the grammar G is:

$$L(G) = \left\{ w_1 | S \Rightarrow^* \begin{bmatrix} w_1 \\ w_2 \end{bmatrix} \in \begin{bmatrix} T^* \\ T^* \end{bmatrix}_\rho \right\}$$

where $\Rightarrow^*$ is a reflexive and transitive closure of $\Rightarrow$.

A **Watson-Crick linear grammar** is a special version of a Watson-Crick context-free grammar where all the rules in the set of rules P are in one of the following forms:

$$A \to \binom{T^*}{T^*} B \binom{T^*}{T^*}, \quad A \to \binom{T^*}{T^*}$$

where $A, B \in N$

A **Watson-Crick regular grammar** is also a special version of a Watson-Crick context-free grammar (and of a Watson-Crick linear grammar) where all the rules in the set of rules P are in one of the following forms:

$$A \rightarrow \binom{T^*}{T^*}B, \quad A \rightarrow \binom{T^*}{T^*}$$

where $A, B \in N$

A further specialization of Watson-Crick regular grammar has been defined in [14] called **1-limited Watson-Crick regular grammar** (N1WK grammar). All rules of such a grammar must contain exactly one terminal symbol on the left-hand side and the starting non-terminal must be the only non-terminal in the grammar. In other words, the form of each rule must be one of the following:

$$S \rightarrow \binom{a}{\lambda}S, \quad S \rightarrow \binom{\lambda}{a}S, \quad S \rightarrow \binom{a}{\lambda}, \quad S \rightarrow \binom{\lambda}{a}$$

where S is the only non-terminal of the grammar.

2.5 Some other models for Watson-Crick languages

This section mentions some other, perhaps slightly less often used, models for Watson-Crick languages — Watson-Crick pushdown automata, Watson-Crick context-free systems and parallel communicating Watson-Crick automata.

2.5.1 Watson-Crick pushdown automata

The Watson-Crick Pushdown automata (WCPDA) have been introduced in [1]. It is basically a two-head pushdown automaton with the complementarity relation added on top. Formally a WCPDA P is a 10-tuple $P = (Q, \#, \$, V, \Gamma, \delta, q_0, Z_0, F, \rho)$ with most symbols having the same standard meaning as in a conventional pushdown automaton — Q is a finite set of states, V is an input alphabet, Γ is a stack alphabet, $q_0 \in Q$ is a starting state, $Z_0 \in \Gamma$ is a starting stack symbol and $F \subseteq Q$ is the set of final states. Symbols $\#, \$ \notin V$ are left and right input markers of the two strands. ρ is the complementarity relation similar to standard WKA.

δ is a set of rules in the following form: $(q, \binom{w_1}{w_2}, x) \rightarrow (q', \gamma)$ where $q, q' \in Q, w_1, w_2 \in V^* \cup \#V^* \cup V^*\$ \cup \#V^*\$, x \in \Gamma, \gamma \in \Gamma^*$. It means that the automaton can transition from state q reading the input w_1 with the first head and w_2 with the second head and go to state q' while removing the top symbol from the stack and putting a string (i.e. 0–n symbols) of the stack symbols onto the stack. The two strands on the input are enclosed in the beginning symbol # and the closing symbol \$, therefore the symbol # may appear in the beginning of w_1 or w_2 and similarly the closing symbol \$ at the end.

A configuration of P is a triple $(q, \binom{x}{y}, \gamma)$ where $q \in Q$ is a state, $\binom{x}{y}$ is the remaining input to be read where $x, y \in \#V^*\$ \cup V^*\$ \cup \lambda$ and $\gamma \in \Gamma^*$ is the content of the stack. The initial configuration of the automaton is $(q_0, \binom{\#w_1\$}{\#w_2\$}, Z_0)$ where $\begin{bmatrix} w_1 \\ w_2 \end{bmatrix} \in WK_\rho(V)$.

A transition $\vdash$ of P is a relation between configurations defined as follows:

$$\left(q, \binom{a_1 w_1}{a_2 w_2}, X\beta\right) \vdash \left(p, \binom{w_1}{w_2}, \alpha\beta\right) \iff \left(q, \binom{a_1}{a_2}, X\right) \rightarrow (p, \alpha) \in \delta$$

$\vdash^*$ is a transitive and reflexive relation of $\vdash$ denoting zero or more transitions.

The language accepted by P is:

$$L(P) = \left\{ w_1 \in V^* | w_2 \in V^* \wedge \begin{bmatrix} w_1 \\ w_2 \end{bmatrix} \in WK_\rho(V) \wedge \left(q_0, \binom{\#w_1\$}{\#w_2\$}, Z_0\right) \vdash^* \left(q, \binom{\lambda}{\lambda}, \alpha\right) \wedge q \in F \wedge \alpha \in \Gamma^* \right\}$$

meaning that there exists a sequence of transitions ($\vdash^*$) from the initial configuration to a final configuration where the remaining input to be read is $\binom{\lambda}{\lambda}$, the content of the stack is arbitrary and the state is in the set of final states F. This means that WK pushdown automata accept by final states.

[1] also defines two special versions of WK pushdown automata — deterministic Watson-Crick pushdown automata (DWKPDA) and strongly deterministic Watson-Crick pushdown automata (SDWKPDA) which are inspired by the deterministic and strongly deterministic WKA.

Watson-Crick pushdown automaton is deterministic if any two rules, which start in the same state, read inputs which are prefix incomparable in the upper or lower part. Formally:

$$\left(q, \binom{u}{v}, X\right) \rightarrow (q', \gamma) \in \delta \wedge \left(q, \binom{u'}{v'}, X\right) \rightarrow (q'', \gamma') \in \delta \Rightarrow u \nsim_p u' \vee v \nsim_p v'$$

Watson-Crick pushdown automaton is strongly deterministic if it is deterministic and its complementarity relation is identity.

2.5.2 Watson-Crick context-free systems

A Watson-Crick context-free systems (WKCFS) have been defined in [15] A WKCFS is: $S = (V, \Sigma, \rho, A, P)$ where V is a finite alphabet, $\Sigma \subset V$, $\rho \subseteq \Sigma \times (V \setminus \Sigma)$ is a complementarity relation where if $(a, \overline{a}) \in \rho$ then $\overline{a} \in V - \Sigma$ is unique for $a \in \Sigma$. A is a finite set of axioms in form $\left[{w \atop s}\right]$ where $w \in \Sigma^*$, $s \in (V - \Sigma)^*$ and $(w, s) \in \rho$. P is a finite set of rules in one of the forms:

$$\binom{a}{\overline{a}} \rightarrow \binom{x}{y}, \;\; \binom{a}{\lambda} \rightarrow \binom{x}{y}, \;\; \binom{\lambda}{a} \rightarrow \binom{x}{y}$$

where $a \in \Sigma$, $(a, \overline{a}) \in \rho$, $x \in \Sigma^*$ and $y \in (V - \Sigma)^*$

A derivation $\Rightarrow$ in S is a relation between $\binom{u_1}{v_1}$ and $\binom{u_2}{v_2}$ defined as follows:

$$\binom{u_1}{v_1} \Rightarrow \binom{u_2}{v_2}$$

if one of the following conditions is met:

1. $\binom{u_1}{v_1} = \binom{x_1}{y_1}\binom{a}{\overline{a}}\binom{x_2}{y_2}$
   $\binom{u_2}{v_2} = \binom{x_1 x x_2}{y_1 y y_2}$ if $\binom{a}{\overline{a}} \rightarrow \binom{x}{y} \in P$
2. $\binom{u_1}{v_1} = \binom{x_1}{y_1}\binom{a}{\lambda}\binom{x_2}{y_2}$
   $\binom{u_2}{v_2} = \binom{x_1 x x_2}{y_1 y y_2}$ if $\binom{a}{\lambda} \rightarrow \binom{x}{y} \in P$
3. $\binom{u_1}{v_1} = \binom{x_1}{y_1}\binom{\lambda}{a}\binom{x_2}{y_2}$
   $\binom{u_2}{v_2} = \binom{x_1 x x_2}{y_1 y y_2}$ if $\binom{\lambda}{a} \rightarrow \binom{x}{y} \in P$

where $a \in \Sigma$, $(a, \overline{a}) \in \rho$, $x_1, x_2 \in \Sigma^*$, $y_1, y_2 \in (V - \Sigma)^*$

The language of S is: $L(S) = \left\{x | \left[{w \atop s}\right] \Rightarrow^* \left[{x \atop y}\right]\right\}$ where $\left[{w \atop s}\right] \in A$, $x \in \Sigma^*$ and $y \in (V - \Sigma)^*$

2.5.3 Parallel communicating Watson-Crick automata systems

Parallel communicating Watson-Crick automata systems (PCWK) have been defined in [3]. PCWK(n) is a PCWK of degree n which is a $(n + 3)$-tuple: $A = (V, \rho, A_1, A_2, ..., A_n, K)$

where V is an input alphabet, ρ is a complementarity relation, $A_i = (V, \rho, Q_i, q_i, F_i, \delta_i)$ for $1 \leq i \leq n$ is a Watson-Crick automaton. $K = K_1, K_2, ..., K_n \subseteq \bigcup_{i=1}^{n} Q_i$ is a set of query states.

The automata A_{1-n} are the components of the system A. A configuration of a PCWK is a $2n$-tuple $\left(s_1, \binom{u_1}{v_1}, s_2, \binom{u_2}{v_2}, ..., s_n, \binom{u_n}{v_n}\right)$ where s_i is a state of component A_i and $\binom{u_i}{v_i}$ is part of the input word that has not yet been read by A_i for $1 \leq i \leq n$.

A transition is a relation $\vdash$ between two configurations and is defined as follows:

$$\left(s_1, \binom{u_1}{v_1}, s_2, \binom{u_2}{v_2}, ..., s_n, \binom{u_n}{v_n}\right) \vdash \left(r_1, \binom{u'_1}{v'_1}, r_2, \binom{u'_2}{v'_2}, ..., r_n, \binom{u'_n}{v'_n}\right)$$

if one of the following conditions is met:

1. $$K \cap \{s_1, s_2, ..., s_n\} = \emptyset \wedge \binom{u_i}{v_i} = \binom{x_i}{y_i}\binom{u'_i}{v'_i} \wedge r_i \in \delta_i\left(s_i, \binom{x_i}{y_i}\right) \text{ for } 1 \leq i \leq n$$

2. for all $1 \leq i \leq n$ such that $s_i = K_{j_i} \wedge s_{j_i} \notin K$ there is $r_i = s_{j_i}$ and for all other $i \leq j \leq n$ there is $r_l = s_l$
   $\binom{u'_i}{v'_i} = \binom{u_i}{v_i}$ for all $1 \leq i \leq n$.

$\vdash^*$ denotes the reflexive and transitive closure of $\vdash$ and the language recognized by PCWK A is:
$L(A) = \left\{w_1 \in V^* | \left(q_1, \begin{bmatrix} w_1 \\ w_2 \end{bmatrix}, q_2, \begin{bmatrix} w_1 \\ w_2 \end{bmatrix}, ..., q_n, \begin{bmatrix} w_1 \\ w_2 \end{bmatrix}\right) \vdash^* \left(s_1, \begin{bmatrix} \lambda \\ \lambda \end{bmatrix}, s_2, \begin{bmatrix} \lambda \\ \lambda \end{bmatrix}, ..., s_n, \begin{bmatrix} \lambda \\ \lambda \end{bmatrix}\right), s_i \in F_i \text{ for } 1 \leq i \leq n\right\}$

2.6 Expressing power of Watson-Crick models

The comparison of expressing power of WK language families in the context of the Chomsky hierarchy has been studied in [10] and [12]. The main result is shown at the figure 2.2. The Chomsky hierarchy is represented on the right (REG — regular languages, LIN — linear languages, CF — context-free languages, CS — context sensitive languages, RE — recursively enumerable languages) while the Watson-Crick languages are on the left. WKREG are languages defined by a non-deterministic Watson-Crick automata or a Watson-Crick regular grammars ([14] shows that these are equivalent). WKLIN are languages defined by WK linear grammars and WKCF are languages defined by WK context-free grammars (it has not been shown, yet, that WK pushdown automata have the same power). The full arrows denote proper inclusion, dotted arrows denote inclusion and dotted lines denote incomparability.

It has been shown in [9] that the type of complementary relation which is used does not increase the expressing power of the Watson-Crick automata and grammars. Also [4] provides an algorithm how to transform any WK automaton to an equivalent WK automaton with the relation being identity. Therefore, many models and algorithms limit themselves to working with identity complementarity relation.

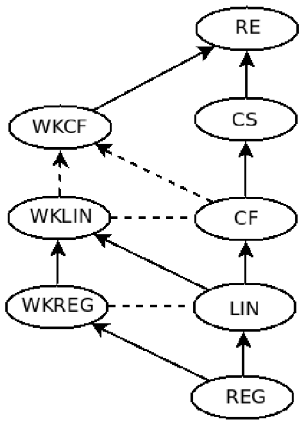

Figure 2.2: Comparison of WK language families in the context of the Chomsky hierarchy

Chapter 3

Existing ways of testing membership in Watson-Crick languages

This chapter focuses mainly on the WK-CYK algorithm which is practically the only algorithm explicitly designed to decide a membership in Watson-Crick languages defined by a WK context-free grammars. Another way could be using WK automata, which is discussed in the latter part.

3.1 The WK-CYK algorithm

The WK-CYK algorithm has been introduced in [11] and it is an enhancement of the CYK algorithm modified for WK context-free languages. To understand it, it is good to be familiar with the way the original CYK algorithm works.

3.1.1 The CYK algorithm as an inspiration of WK-CYK

The CYK algorithm is named after J. Cocke, D. Younger and T. Kasami [2], [17] [8]. It is used to decide the membership in a language defined by a context-free grammar, which must be in the Chomsky normal form (CNF).

On the input there is a string and a grammar and the algorithm decides whether the string belongs to the language defined by the grammar (accepts or rejects the string). There are two kinds of rules in a grammar in the CNF (disregarding the $S \to \epsilon$ rule which is used only to include empty string in the language): $A \to a$ and $A \to BC$ where A, B, C are non-terminals and a is a terminal.

In the first stage, it analyzes the first kind of rules — each of the symbols from the input string has to be generated by a rule or several rules of this form. Thus, it gets a set of candidate non-terminals for each symbol.

In the next stage it uses the second kind of rules. Every non-terminal (except the starting one) has to be generated by such a rule. The algorithm is looking for rules which can generate the candidate non-terminals which have been found in the previous stage. All possible combinations need to be considered, for instance the sequence of non-terminals ABC may be generated by rules $X \to AB$ and $Y \to XC$ or by rules $X \to BC$ and $Y \to AX$. In this way, the algorithm needs to find all possible ways to generate words of increasing length (all parse trees). Finally, it needs to find a non-terminal that can generate the whole word and, at the same time, it must be the starting non-terminal in the given grammar. If it succeeds, the word given on the input is in the language, otherwise it is not.

The complexity of the CYK algorithm is $O(n^3 \times R)$ where n is the input string length and R is the number of rules in the grammar.

3.1.2 Watson-Crick Chomsky normal form

Just like the CYK algorithm works with grammars in the Chomsky normal form, the WK-CYK algorithm requires grammars to be in the Watson-Crick Chomsky normal form. The Watson-Crick Chomsky normal form (WK-CNF) is a modification of CNF for Watson-Crick context-free grammars. A grammar in the WK-CNF has only rules of one of the following forms:

- $A \to \binom{a}{\lambda}$
- $A \to \binom{\lambda}{a}$
- $A \to BC$
- $S \to \binom{\lambda}{\lambda}$ (this rule is used only to include an empty word in the language)

where A, B and C are non-terminals, S is the starting non-terminal and a is a terminal of the grammar. It is possible to transform any WK context-free grammar to the WK-CNF. The steps are mostly analogous to the transformation of the standard context-free grammar to the CNF. This process includes:

1. removing λ-rules (rules of the form $A \to \binom{\lambda}{\lambda}$)
2. removing unit rules (rules of the form $A \to B$)
3. removing useless rules and symbols (symbols that are unreachable from the starting symbol or cannot lead to a terminal string and rules which use such symbols)
4. replacing every terminal on the left-hand side of each rule (except the rules already in the right form) with a new non-terminal and adding a new corresponding rule
5. breaking down the rules producing non-terminals, so that they produce only two at a time

The procedure of the transformation is described formally in [11].

3.1.3 Order of generating terminals in WK grammar

A complication compared to the CYK algorithm that WK-CYK has to deal with, is the ambiguity in the order of generating terminals. In case of a standard context-free grammar in the CNF, the order of non-terminals that generate a word, for instance *abcd*, is clear — if the rules are $A \to a$, $B \to b$, $C \to c$ and $D \to d$, the non-terminal word that generates the terminal string *abcd* must be $ABCD$. The order cannot change.

But in case of WK grammars, the order is not clear. For the terminal string $\binom{ab}{cd}$, the only given order of generation is a before b and c before d, anything else is uncertain. If the rules are $A \to \binom{a}{\lambda}$, $B \to \binom{b}{\lambda}$, $C \to \binom{\lambda}{c}$ and $D \to \binom{\lambda}{d}$, that terminal word can be produced by six different orderings of the non-terminals: $ABCD$, $ACBD$, $ACDB$, $CABD$, $CADB$ and $CDAB$.

3.1.4 Description of the WK-CYK algorithm

The WK-CYK algorithm taken from [11] is on the figures 3.1 and 3.2.

```
procedure SetsConstruction:
Input: string [w/w] = [$x_{11}x_{12}...x_{1n}/x_{21}x_{22}...x_{2n}$]

for 1 $\leq i \leq n$ do
    $X_{i:i,0:0} = \{A : A \rightarrow (x_{1i}/\lambda)\}$
    $X_{0:0,i:i} = \{A : A \rightarrow (\lambda/x_{2i})\}$

for $2 \leq y \leq 2n$ do
    for $0 \leq \beta \leq n$ do
        $\alpha = y - \beta$
        if $\alpha = 0$ then
            i = j = 0
            for $1 \leq k \leq n - y + 1$ do
                l = k + y - 1
                ComputeSet $X_{i:j,k:l}$
        else if $\beta = 0$ then
            k = l = 0
            for $1 \leq i \leq n - y + 1$ do
                j = i + y - 1
                ComputeSet $X_{i:j,k:l}$
        else
            for $1 \leq i \leq n - \alpha + 1$ do
                for $1 \leq k \leq n - \beta + 1$ do
                    j = i + $\alpha$ - 1
                    l = k + $\beta$ - 1
                    ComputeSet $X_{i:j,k:l}$
if $S \in X_{1:n,1:n}$ then
    w ∈ L(G)
else
    w ∉ L(G)
```

Listing 3.1: Procedure SetsConstruction of WK-CYK

```
procedure ComputeSet:

if i = j = 0 then
    $X_{0:0,k:l} = \left\{\bigcup_{t\in[k,l-1]} X_{0:0,k:t}X_{0:0,t+1:l}\right\}$
else if k = l = 0 then
    $X_{i:j,0:0} = \left\{\bigcup_{s\in[i,j-1]} X_{i:s,0:0}X_{s+1:j,0:0}\right\}$
else
    $X_{i:j,k:l} = \left\{X_{i:j,0:0}X_{0:0,k:l} \cup X_{0:0,k:l}X_{i:j,0:0}\right\}\cup$
        $\bigcup_{s\in[i,j-1],t\in[k,l-1]} \left\{X_{i:s,k:t}X_{s+1:j,t+1:l}\right\}\cup$
        $\bigcup_{s\in[i,j-1]} \left\{X_{i:s,k:l}X_{s+1:j,0:0} \cup X_{i:s,0:0}X_{s+1:j,k:l}\right\}\cup$
        $\bigcup_{t\in[k,l-1]} \left\{X_{i:j,k:t}X_{0:0,t+1:l} \cup X_{0:0,k:t}X_{i:j,t+1:l}\right\}$
```

Listing 3.2: Procedure ComputeSet of WK-CYK

WK-CYK algorithm expects a grammar in the WK-CNF and a double stranded string on the input. Since one of the algorithm's requirements is that the complementarity relation must be identity, the upper and lower strands are always the same.

WK-CYK uses sets marked as $X_{a:b,c:d}$. These are sets of non-terminals that can generate a segment of the input double stranded string specified by the indexes a, b, c and d. a and b are indexes of terminals in the upper strand and specify an interval (the indexing starts with the index 1 and the edge indexes are included). The lower strand interval is specified by indexes c and d. If a pair of indexes is 0, no symbols from the corresponding strand are included. For instance, for the segment $\binom{abcd}{abcd}$, $X_{2:2,0:0}$ would contain a set of non-terminals that generate $\binom{b}{\lambda}$, $X_{2:4,1:3}$ non-terminals that generate $\binom{bcd}{abc}$.

The main procedure of WK-CYK

In the first step (lines 4–6 of figure 3.1) WK-CYK finds sets $X_{i:i,0:0}$ and $X_{0:0,i:i}$ for $0 < i \leq |n|$ (n is the length of the input). These are non-terminals that directly generate single terminals. Then, it searches for ways to generate segments of the input of increasing lengths, beginning with length of 2 and up to the length of $2n$. It is because in the input of length n there are actually $2n$ of terminals — n in the upper and n in the lower strand. For each length of the segment it takes all possible combinations of number of symbols from the upper and the lower strands. For instance, if the length of the current segment is 3, that can include 3 terminals from the upper strand and 0 from the lower or 2 and 1, 1 and 2, 0 and 3.

For each of these segments, it calls the procedure *ComputeSet* which finds all non-terminals, that could generate the given segment. When WK-CYK uses this procedure to compute set X of a segment of length n, it is necessary to have already computed sets X for all segments of length $m < n$. Therefore it proceeds from the length 1 upward.

Let us consider an example with input $\binom{abcd}{abcd}$. The first step looks for way of generating the individual terminals, in other words non-terminals that generate $\binom{a}{\lambda}$, $\binom{\lambda}{a}$, $\binom{b}{\lambda}$ etc. Then it looks for non-terminals that could generate two terminals, meaning either two terminals from the upper strand, two terminals from the lower strand or one from each. In each of these cases, all possible combinations need to be considered. If the two terminals are from the upper strand, the combinations are either $\binom{ab}{\lambda}$, $\binom{bc}{\lambda}$ or $\binom{cd}{\lambda}$ (or using the X sets: $X_{1:2,0:0}$, $X_{2:3,0:0}$ or $X_{3:4,0:0}$). It the two terminals are one from each strand, there are 16 combinations $\binom{x}{y}$ where $x, y \in \{a, b, c, d\}$ ($X_{1:1,1:1}$, $X_{2:2,1:1}$, $X_{1:1,2:2}$ etc.). And for terminals from the lower strand, the combinations are $\binom{\lambda}{ab}$, $\binom{\lambda}{bc}$ or $\binom{\lambda}{cd}$ ($X_{0:0,1:2}$, $X_{0:0,2:3}$ and $X_{0:0,3:4}$).

When the segment of length $2n$ has been computed, WK-CYK is finished. It has succeeded if the starting symbol S can generate the whole input, in other words: if $S \in X_{1:n,1:n}$.

The ComputeSet procedure

The *ComputeSet* procedure has as a parameter a segment of input specified by the four indexes. It searches all pairs of sets X which could together produce the given segment. If the segment consists of symbols from one strand only ($X_{i:j,0:0}$ or $X_{0:0,i:j}$), the situation is simpler — it needs to consider the pairs of sets X that produce the segment split in any two parts. If the segment contains symbols from both strands, there are more ways to split it:

- The first set could produce the entire upper strand and the second set could produce the entire lower strand or the other way around (the order matters)
- The first set could produce the entire upper (or lower) strand and any part the lower (upper), the second one would produce the rest of the divided strand. Again, all possible divisions of the divided strand need to be considered.
- Both sets could produce parts of both strands. Again, all possible combinations of divisions of both strands need to be considered.

When all combinations of two X sets potentially producing the input segment have been found, the procedure then needs to check the grammar rules to find those rules which actually generate non-terminal from these sets. This step is not explicitly described in the *ComputeSet* procedure. For each such a rule, the non-terminal on its left-hand side is going to be included in the procedure's result. The final result is then a set of all such non-terminals.

Lets us consider an example, where the procedure computes $X_{1:2,1:2}$, in other words, the segment $\binom{ab}{ab}$. All possible divisions of this segment are in the table 3.1:

Figure 3.1: All possible divisions of the segment $\binom{ab}{ab}$

	sub-segments	corresponding sets	example of sets contents
1.	$\binom{ab}{\lambda}$, $\binom{\lambda}{ab}$	$X_{1:2,0:0}$, $X_{0:0,1:2}$	$\{N_1\}$, $\{N_2\}$
2.	$\binom{\lambda}{ab}$, $\binom{ab}{\lambda}$	$X_{0:0,1:2}$, $X_{1:2,0:0}$	$\{N_3\}$, $\{N_4, N_5\}$
3.	$\binom{ab}{a}$, $\binom{\lambda}{b}$	$X_{1:2,1:1}$, $X_{0:0,2:2}$	$\{N_6\}$, $\emptyset$
4.	$\binom{a}{ab}$, $\binom{b}{\lambda}$	$X_{1:1,1:2}$, $X_{2:2,0:0}$	$\emptyset$, $\emptyset$
5.	$\binom{a}{\lambda}$, $\binom{b}{ab}$	$X_{1:1,0:0}$, $X_{2:2,1:2}$	$\emptyset$, $\emptyset$
6.	$\binom{\lambda}{a}$, $\binom{ab}{b}$	$X_{0:0,1:1}$, $X_{1:2,2:2}$	$\emptyset$, $\emptyset$
7.	$\binom{a}{a}$, $\binom{b}{b}$	$X_{1:1,1:1}$, $X_{2:2,2:2}$	$\emptyset$, $\emptyset$

All the 14 X sets from the middle column must already be computed, they are sets for segments of lengths 1, 2 and 3. Each of these sets contains zero or more non-terminals that can produce the given sub-segment. In last step, the procedure checks all rules of the grammar of type $A \to BC$ to find rules where B is in the first X set and C is in the second X set of one of the divisions of the segment. In the example, there are three combinations of non-terminals that could lead to a result: N_1N_2, N_3N_4 and N_3N_5 (N_6 is alone — that is not not enough to produce the segment). Therefore, the result will be a set $\{X, Y, Z\}$ if there are rules $X \to N_1N_2$, $Y \to N_3N_4$ and $Z \to N_3N_5$. If only subset of these rules is found, the resulting set will contain only left-hand sides of those rules (or could even be an empty set).

Two remarks regarding WK-CYK

1. The loop on the line 9 of the procedure *SetsConstruction* (figure 3.1) iterates β from 0 to n. In this context, β represents the length of the lower strand segment while α represents the length of the upper strand segment and $\alpha = y - \beta$ where y is the length of the whole

segment. Part of the loop actually calculates with non-sensical values. When calculating with segment that is shorter then one strand (i.e. $y < n$), it includes the case when $\beta > y$ and so $\alpha < 0$. In other words, the algorithm splits the segment of length, for instance, 2, to two parts of lengths 3 and -1.

When calculating with segment that is longer than the input, it includes the case where β is too short and so α is then longer then a strand length. In other words, if the input length is, for instance, 4 and the segment length is 8, it splits the segment to lengths 7 and 1 which is not possible with the input length of 4.

This does not affect the correctness of the computation because the non-sensical values find no result. However, more precise and efficient solution would be to iterate β over the interval: $\langle max(y - n, 0), min(n, y)\rangle$ instead of interval $\langle 0, n\rangle$.

2. The time complexity of WK-CYK is $\mathcal{O}(n^6)$. As described in [11] (section 6), the WK-CYK main procedure has complexity of $\mathcal{O}(n^4)$ and the nested procedure *ComputeSet* has complexity of $\mathcal{O}(n^2)$. This is true with respect to the input length. Possibly, more precise description of the complexity would be $\mathcal{O}(n^6 \times R)$ where n is the input length and R is the number of rules in the grammar. The description of the procedure *ComputeSet* uses the operation of set union ($\cup$), as if it has constant time complexity which, in reality, it does not — it requires iterating over the rules of the grammar.

3.2 Using automata to test the membership in Watson-Crick languages

In general, automata seem more suitable for deciding the membership in a language than grammars. In case of deterministic automata, the situation is quite straightforward — every input deterministically leads to the next state and when the whole input is read, the automaton decides whether the input is accepted by finishing in a final state or not. The situation is not as clear in case of the weaker types of deterministic automata.

3.2.1 Strongly deterministic Watson-Crick automata

The strongly deterministic automata represent the simplest case as they have clearly linear complexity. A disadvantage may be the fact that strongly deterministic automata are weaker then deterministic automata.

3.2.2 Weakly deterministic and deterministic Watson-Crick automata

Weakly deterministic and deterministic automata are similar in the sense that their determinism depends on the configuration being known. But the membership of a string in a WK language is defined by the string being equal to upper strand only (see section 2.2). A corresponding lower strand simply needs to exist but is not automatically known in advance. If this is not the case and the entire input with both strands is available at the beginning, then there is practically no difference between the three types of deterministic WKA. The strongly deterministic WKA knows the lower strand thanks to the fact that the complementarity relation is identity — the strands must be identical.

If the lower strand is not known, however, the deterministic WK automata are in practice the same as non-deterministic WK automata. It is, in general, not necessarily clear what the next state will be given the input. This is demonstrated by example on the left on the figure 3.2. The snippet of the WK automaton has two transition rules: $q_0\binom{a}{b} \rightarrow q_1$ and

$q_0\binom{a}{c} \rightarrow q_2$. Let us suppose that both $(a, b) \in \rho$ and $(a, c) \in \rho$ and there is the string aaa on the input. These rules fulfill the condition for deterministic WKA which, for this case, is: $a \not\sim a \vee b \not\sim c$. It is not possible to choose the next path based on the upper strand as a is a prefix of a. However, it should be possible to decide based on the lower strand — either a symbol b is going to be read and the automaton will transition to q_1 or symbol c and the automaton will transition to q_2. But if the process of accepting a string is at the same time looking for a suitable lower strand, it is up to the automaton to decide between these two possibilities and produce either the symbol b to the lower strand or the symbol c.

The figure 3.2 on the right shows a snippet of a completely non-deterministic WKA. The rules shown are: $q_0\binom{a}{b} \rightarrow q_1$ and $q_0\binom{aa}{bb} \rightarrow q_2$. Even if both input strands are known, in case the upper strand contains aa and the lower strand contains bb, both paths are possible. Either one symbol from each strand will be read and transition will lead to q_1 or two symbols from each strand will be read and the transition will lead to q_2. So in practice, there is not much difference between a weakly deterministic WKA, deterministic WKA and non-deterministic WKA unless the lower strand is known in advance.

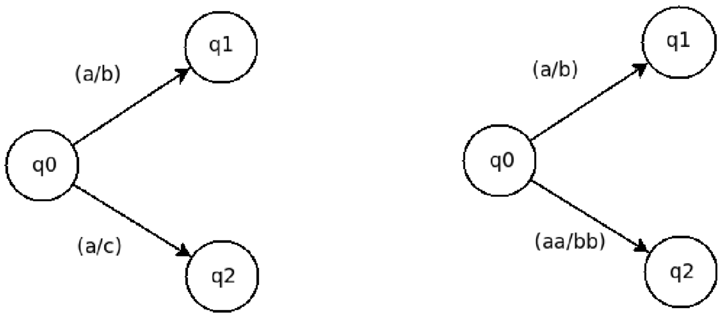

Figure 3.2: Example of non-determinisms in WKA

3.2.3 Accepting inputs by a non-deterministic Watson-Crick automaton

It is possible to propose a similar algorithm for a run of WKA as for non-deterministic finite automata (FA). Such an algorithm for a finite automaton needs to consider all possible paths the FA may take — in practice it must remember, not just one state where the FA is at the moment, but a subset of all its states. However, this is not enough in case of WKA. An algorithm which controls a run of a non-deterministic WKA needs to remember not only all the states where the automaton can be at a given moment, but also, what inputs have been read from the upper strand and what inputs have been generated to the lower strand. The figure 3.3 demonstrates the difference between undeterminism in run of a FA and WKA.

The FA snippet on the left starts in state q_0. If the next symbol on the input is a, the automaton can transition to state q_1 or q_2. This means that the algorithm keeps in mind that the current state is one of the two (a subset of all FA states: $\{q_1, q_2\}$). If there is another a which causes another non-deterministic choice, it turns out that the number of possibilities can even decrease or, as in this case, stay the same because some branches are merged. The new state is either q_3 or q_4.

The situation in case of WKA on the right is much more complicated. Even though it may seem like the paths merge in the state q_4, they do not. Let us assume that the input string is aaa and that $(a, b) \in \rho$ and $(a, c) \in \rho$. The automaton has three choices:

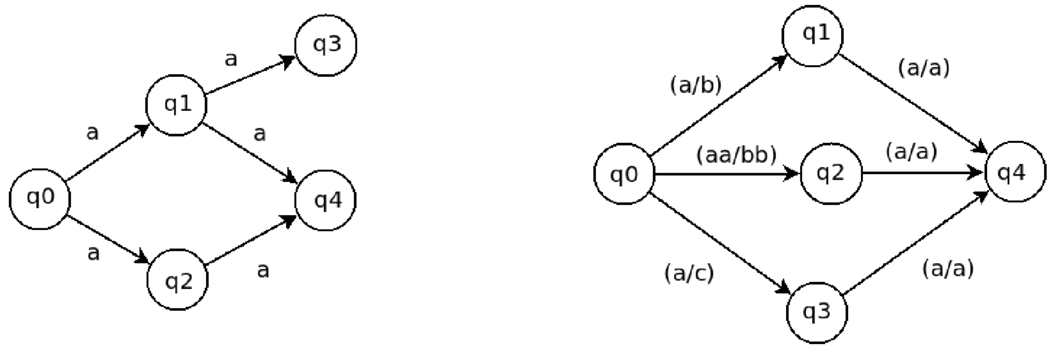

Figure 3.3: Non-determinisms in a WKA and a finite automaton

1. Transition to q_1, read symbol a from the input and generate symbol b to the lower strand. This may be possible even if $(a, b) \notin \rho$ because a different number of symbols might have been read from the two strands (they are, in a way, out of sync).

2. Transition to q_2, read two symbols aa and generate two symbols bb to the lower strand.

3. Transition to q_3, read symbol a and generate symbol c.

In these three cases, there are more differences between these states of the algorithm then just the the different WKA state. Therefore, after the next step, the three paths will still differ in what has been read from the input (upper strand) and what has been generated to the lower strand.

The algorithm that would simulate the run of a WKA could therefore be designed as a state space searching algorithm. Each node would contain three items — 1. what state is the WKA in, 2. what remains to be read from the input, 3. what has been generated to the lower strand. The solution would then be a node where WKA is in a final state, nothing remains to be read and the two strands (one read, one generated) adhere to the complementarity relation.

This is quite close to the solution that is proposed in the following chapter with the main difference that the space state search is based on WK context-free grammars instead of automata as they have more expressive power (see 2.6).

Chapter 4

Testing membership by searching the state space

This chapter introduces the main algorithm of this thesis for testing membership in WK context-free languages. In this thesis it is referred to as the **state space search** or the **tree search**. Its core is a standard Breadth-first search algorithm (BFS) with various optimizations added on top.

Standard BFS starts with a root node. In case of grammars, that is the starting non-terminal symbol of the grammar. Then the tree is built by applying all possible rules to all possible non-terminals. Each rule application generates a new node. The node contains a word which consists of some non-terminals, some terminals in the upper strand and some terminals in the lower strand.

The BFS algorithm always finds a solution if there is one. It finds the optimal solution which, in this case, means the shortest sequence of rules that generate the input string from the starting non-terminal. However, whether the solution is optimal or not is irrelevant for the membership problem. If there is no solution, the algorithm will probably never stop, as the state tree is usually infinite. Also, such a tree would grow very rapidly and the solution would usually not be found in a reasonable time frame. Therefore, some optimizations need to be used. This work introduces two key kinds of optimizations. Firstly, identifying dead ends in the search tree and removing them from the computation — this is referred to as **pruning**. Secondly, choosing such nodes for the subsequent computation which seem to be the most promising in leading to the solution. This is referred to as **node precedence**.

4.1 Key characteristics of the state space search

Besides pruning and node precedence heuristics, the algorithm keeps a set of states which have been generated (added to the tree), in order to avoid analyzing the same word repeatedly or even getting stuck in a loop. Also, it considers leftmost derivation only. This means that a node which contains several non-terminals can generate new nodes only by applying rules to the first non-terminal in the word.

The figure 4.1 shows an example of a tree search progress. The rules of its grammar in this example are $S \to SS \mid ABC$, $A \to \binom{a}{a} \mid \binom{b}{b}$ and some rules $B \to ...$, $C \to ...$ which are not important. S is the starting non-terminal, therefore, S is the first node and there are two possible rules that can be applied to S, so this node has two successors. The node precedence heuristic will choose one of the successors to be analyzed next — perhaps the left one with word ABC. This node, too, only has two successors, which are made by the two rules that can be applied to the first non-terminal — A. Even though there are some rules for B and C, these rules are not used to produce successors, yet. The nodes created by rules applied on B would be successors of the words $\binom{a}{a}BC$ and $\binom{b}{b}BC$ which have the symbol B as the first non-terminal from the left.

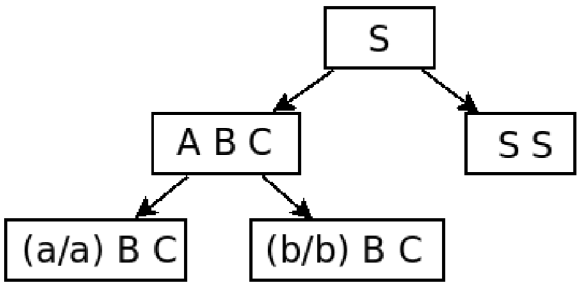

Figure 4.1: Example of a search tree

In a word of a WK grammar, the terminals are clustered together into segments. If two segments appear next to each other in a word, they are merged. These segments, as well as non-terminals, are referred to as **letters** because together they constitute words. For instance, a three-letter word: $\binom{abc}{\lambda}A\binom{b}{b}$ after application of rule $A \to \binom{\lambda}{a}$, will result in a word with just one letter: $\binom{abcb}{ab}$.

The word in a node that is the solution needs to meet the following criteria:

1. It contains no non-terminals. Since neighboring terminal segments are always merged this implies that there is only one letter — a segment of terminals.
2. The upper and lower strands of this segment are of the same length.
3. Each pair of symbols from the upper and lower strands with the same index must be related by the complementarity relation.
4. The upper strand must be equal to the input string.

If the criteria are met, the algorithm has found the right node and that means the input string is a part of the language defined by the grammar. It has been accepted by the state space search algorithm. If the whole state space has been searched (in case it is not infinite), there is no solution and the input has been rejected by the state space search algorithm.

4.2 Identifying a dead end in the state tree

A blind BFS would stop searching a branch only when all non-terminals have been used to generate all possible terminal words (words with terminals only). But sometimes it is possible to tell in advance that a specific word cannot lead to the desired solution. If that is the case, the node can simply be removed and the whole branch which it would generate is skipped. The next section describes various ways (heuristics) of recognizing the dead branches. These are referred to as pruning heuristics, there are five of them and each one has an abbreviation which is used further on.

1. Detecting that one of the strands is already too long — SL

2. Detecting that the overall word is already too long — TL
3. Matching the starting terminals in the upper strand to the input — WS
4. Checking the complementarity relation — RL
5. Comparing the input to a regular expression generated from the word — RE

4.2.1 One of the strands is too long (SL)

A terminal symbol which appears both in the upper or lower strand can never disappear further in the branch. That means that the count of all symbols in upper and in the lower strand must not be grater then the length of the input string. Otherwise the solution can never be reached from that branch.

4.2.2 The word including non-terminals is too long (TL)

Non-terminals present a more complex problem when dealing with the length of the word. First of all, the algorithm calculates in advance how many terminals each non-terminal produces at minimum. For instance, if the grammar contains rules: $A \rightarrow AA \mid \binom{ab}{cd} \mid BB$ and $B \rightarrow \binom{a}{\lambda}$, the non-terminal B produces always one terminal, that means one terminal at minimum. The non-terminal A can produce various number of terminals, but two at minimum — thanks to the rule $A \rightarrow BB$ and the fact that B has the minimum of one. This value is than considered to be the length of the given non-terminal. This length can be applied both to the upper or to the lower strand because, in general, it is not known which strand will absorb the symbols generated from the non-terminal. This then leads to the following constraint on the word:

$$|upper| + |lower| + |nts| \leq 2 \times |input|$$

where $|upper|$ and $|lower|$ are the counts of terminals in the upper and lower strands, $|nts|$ is the length of all non-terminals in the word and $|input|$ is the length of the input string. If this constraint is broken, the word cannot lead to the solution and the branch can be pruned.

If the grammar contains no λ-rule (rule of the form $N \rightarrow \binom{\lambda}{\lambda}$), This constraint guarantees that the algorithm will finish. Once all the words within the given length limit have been generated and a solution not found, the search will end.

If the grammar does contain λ-rules, the previous constraint can still be applied — the non-terminals that can be erased are assigned the length of zero. In this case, it is not possible to guarantee that the search will end, because the non-terminals of length zero can be combined infinitely many times. However, it is possible to utilize the algorithm for removing λ-rules (which is described in [11] and which is implemented in the application described in the next chapter).

4.2.3 The beginning of the word does not match the input (WS)

If a word in a node begins with some terminal symbols in the upper strand, these symbols will always stay at the beginning further in the given branch. Unlike the other terminals, these starting terminals already have fixed indexes. If these symbols do not match the prefix of the input string of the same length, the input string can never be generated from this branch. If on the other hand, the word starts with a non-terminal, there is nothing to be said about what can be at the beginning of the word further in the branch.

It is possible to check the end of the word in the same manner but the generation is performed from the left to the right and so there is little benefit in checking the end.

4.2.4 Checking the complementarity relation (RL)

As previously described, the symbols in the upper and lower strands with the same indexes must be related by the complementarity relation. Unfortunately, this can be checked only at the beginning of the word (Technically, it can be checked at the end as well, while indexes of these symbols are not yet known, the last terminal symbol will always stay the last. But just like in the case of previous heuristic, there is little benefit in checking the end when the generation is done from the left side.). Indexes of the symbols in the middle part (anywhere after the first non-terminal) are not known. Thus this check can be understood as an extension of the previous one — if the word begins with some terminal symbols and there are some symbols in both the upper and lower strands, these symbols can be tested whether they adhere to the complementarity relation. But only to the length of the shorter of the two strands in this letter.

4.2.5 The input matches a regular expression generated from the word (RE)

It is possible to generate a regular expression that represents the current word. Each non-terminal serves as a wild card (`.*`). Each terminal in the upper strand stands for itself. Lower strand is ignored. This expression must be matchable to the input string, otherwise it is not possible to generate it from the current branch. For instance, if the word is

$$\binom{abc}{f} N_1 \binom{d}{gh} N_2 \binom{e}{i} N_3$$

then the resulting regular expression will be: `^abc.*d.*e`. The symbols *abc* must be at the beginning (therefore the `^` denoting the beginning of the expression is placed at the start); then it is not known what will be generated by the non-terminal N_1 — therefore the wildcard is there next; then there will have to be a symbol d; another wildcard for non-terminal N_2; symbol e; and then anything. The regular expression might end with a wildcard generated from the last non-terminal N_3 but that is not necessary. Wildcard before and after the expression is implicit. A starting non-terminal can be represented by omitting the symbol `^` which denotes the beginning of the string. An ending non-terminal can be represented by omitting the symbol `$` which denotes the end of the string.

The order in which the pruning heuristics are applied matters. It is good to first apply the heuristics that are more likely to succeed and that require less computational power. If they are successful, the more complex heuristic can be skipped.

It is possible to come up with some more checks that could identify a dead end in the search tree. The disadvantage of any check is the computing power that has to be used for checking any node that is generated and analyzed. If some checks are unlikely to significantly prune the tree and/or are complicated to compute, it is not clear if they will improve the actual performance of the algorithm.

4.2.6 Examples of pruning

Let us consider a grammar with no λ-rules with the identity complementarity relation, the input string *abcd* and the following words:

1.
$$\binom{a}{ab} N_1 \binom{\lambda}{cd} N_2 \binom{\lambda}{e}$$
The input string can never be generated from this word because the fragments of the lower strand are too long already — it has five symbols and the input string only has four. The SL pruning will remove this word.

2.
$$\binom{ab}{ab} N_1 N_2 \binom{\lambda}{d} N_3 N_4$$
This word would be promising if there had been some λ-rules. Since there are not, the word contains too many non-terminals. Each of them is going to generate at least one terminal symbol and only three symbols are missing (c and d in the upper strand and c in the lower strand). Inevitably, there will be at least one symbol too many. This word will be removed by the TL pruning.

3.
$$\binom{abd}{\lambda} N_1 \binom{\lambda}{ab} N_2$$
Regardless of what can be generated from N_1 and N_2, the upper strand will always have to begin with symbols *abd*. There is no way how to insert c between b and d. Therefore the input string can never be generated from here and this word will be removed by the WS pruning.

4.
$$\binom{abc}{ac} N_1 \binom{\lambda}{d} N_2$$
The upper strand looks promising as it starts with the same symbols *abc* as the input string. But the lower strand starts with *ac*. The first symbol pair (a/a) adheres to the complementarity relation, the second one (b/c) does not. The third symbol in the upper strand — c cannot be related to any symbol — it has no counterpart, yet. The check was always going to end with the second symbol pair. Anyway, this word will be removed by the pruning RL.

5.
$$N_1 \binom{b}{\lambda} N_2 \binom{a}{\lambda} N_3$$
Whatever is generated from the non-terminals N_1, N_2 and N_3, the upper strand will always keep the order of symbols — first, the symbol b, then the symbol a (with potentially some symbols before, in between and after). That can never result in the string *abcd*. This word will be removed by the pruning RE.

4.3 Heuristics for node precedence

The aim of the node precedence heuristics is to choose a path in the search tree, which is likely to lead to the solution, the more promising nodes are taken before the others and their successors are generated sooner. The individual heuristic functions attempts to answer the question — which node is more promising than the rest? It assigns each node a number — an evaluation of the node. The lower the node evaluation, the higher priority the node has.

Such heuristics can only be effective if the answer of the search is positive — if there actually is a solution. Unfortunately, if it is negative, it does not help that the algorithm eliminates the more promising branches of the tree first. Eventually, it will have to search through all possible states anyway, in order to make sure that there is no solution.

The following node precedence heuristics have been implemented. Each heuristic also has an abbreviation that will refer to it further on.

- No heuristic (NONE) — the evaluation of the word is always 0. This is used for comparison to the other heuristics.
- Aversion to non-terminals (NTA) — the evaluation is equal to the count of non-terminals in the word
- Weighted aversion to non-terminals (WNTA) — each non-terminal has a pre-calculated weight, which is the minimum amount of rules that must be used in order to generate a terminal word from it. The evaluation is equal to the sum of the weights of all non-terminals in the word.
- The terminal matching — there are three variants that differ slightly (TM1, TM2, TM3). Each of them increases the priority (i.e. decreases the evaluation) for each upper strand non-terminal (going from left to right) which matches the input string symbol on the same index.
  - TM1 examines terminals going from the left while ignoring non-terminals, decreases evaluation (i.e. increases priority) for each match and finishes when it discovers the first difference.
  - TM2 is similar to TM1, but when it discovers a difference, it does not finish but increases the evaluation and moves on
  - TM3 evaluates the first item in the word only. If it is a non-terminal, it returns zero.
- Combinations of NTA/WNTA and TM1/TM2/TM3 — There are six combinations because it does not make sense to combine NTA and WNTA or TM1-3 together: NTA+TM1, NTA+TM2, NTA+TM3, WNTA+TM1, WNTA+TM2, WNTA+TM3.

In summary, there are 12 node precedence heuristics considered in total (including the first, empty heuristic). Unlike in case of pruning, where all methods can be applied at the same time, there can be only one node precedence heuristic active at one time. Therefore chapter 6 contains the tests and comparison of the effectiveness of these heuristics.

4.4 Theoretical complexity of the state space search

The state space search algorithm uses Breadth-first search (BFS) as its basis. Both the time and space complexity of BFS are $\mathcal{O}(b^d)$ where b is the maximum number of successors of a node (branching factor) and d is the depth of the tree. The branching factor is then equal to the maximum number of rules of the given grammar that have the same non-terminal on the left-hand side. This is because always only the first non-terminal in the word is used to generate successors in the tree. The depth of the tree is going to be different for different grammars and even for different inputs.

In general, the theoretical complexity of the state space search algorithm is not impressive, it is much worse then WK-CYK's $\mathcal{O}(n^6)$ or $\mathcal{O}(n^6 \times R)$. However, this is because it has been designed with a rather practical approach, it relies heavily on the heuristics and optimizations and so its performance is usually much better.

4.5 Parallelization of the state space search

The parallelization of the state space search should be very much possible and straightforward. The algorithm uses a priority queue to store all the nodes which are to be analyzed. Therefore, multiple processes could be taking nodes from the queue and analyze them independently. There are many variants how this could be done — analyzing one node at the time and returning result immediately to the queue shared by all process or analyzing independently whole segments of the tree with less need for synchronization but, perhaps, more redundant work — these are questions of the efficiency of the actual implementation which is out of the scope of this work.

Chapter 5

Implementation of the state space search

The implementation has been done in the language Python (version 3.9.7). The main components are the following:

- the class representing a Watson-Crick context-free grammar, including the implementation of the WK-CYK algorithm and the state space search algorithm
- classes representing a rule of a grammar and a tree node
- a set of grammar definitions and generators of the input strings
- a set of test scripts that run the various tests comparing heuristics, performance etc.
- test runner class which is a middle layer between the test scripts and the main grammar class.

In the code and its description I use the term **word** to refer to the right hand side of the rules and, in general, a list of letters. The term **letter** is a one element of the word, which is either a non-terminal, or a segment with terminals. Such segments with terminals are stored as a pair (tuple) of two lists — upper and lower strand. For instance, a word $A\binom{abc}{\lambda}B$ has three letters: non-terminals A and B and a segment of non-terminals $\binom{abc}{\lambda}$, which contains two lists, first (upper strand) with three items — terminals a, b and c and the second one (lower strand) is an empty list.

5.1 Implementation of the main class representing the grammar

The class representing a grammar is called *cWK_CFG*, it contains the following data:

- the items which define the grammar: *nts* — the set of non-terminals, which are represented by alpha-numeric characters; *ts* — set of terminals, also represented by alphanumeric characters; *startSymbol* — starting non-terminal; *rules* — set of grammar rules, which are objects of the class *cRule*; *relation* — list representing a complementarity relation, it contains tuples of two terminals. These five items are parameters which need to be passed to the class constructor. This corresponds to the way how a context-free WK grammar is formally defined.

- *nodePrecedenceList* — a list of all implemented node precedence heuristics, it is stored as a list of pairs (tuples) of heuristic name and function; *currentNodePrecedence* is the index of the active one

- *pruningOptions* — a dictionary with a pruning function as a key and a boolean as a value indicating which pruning heuristics are active; *pruneCnts* — a dictionary of pruning functions as keys and integer values which counts, how many times the given pruning has been successfully used

- *ruleDict* — grammar rules stored in a dictionary with non-terminals as keys and list of rules as values. This is a more efficient way of accessing rules for a given non-terminal then iterating over all rules and filtering them based on left-hand side non-terminal.

- *relDict* — complementarity relation stored in a more convenient way in a dictionary with first symbol as a key and string of symbols (symbols that are related to the key symbol) as a value. This is more efficient way of finding all symbols related to a given symbol.

- *ntDistances* — a pre-calculated dictionary, with all non-terminals as keys and distance to terminals as values. This distance is a minimum number of rules which lead from the given non-terminal to a word with terminals only. This is used for the node precedence heuristic WNTA.

- *erasableNts* — a pre-calculated set of non-terminals which can be erased by applying certain sequence of rules. It is used for removing the λ-rules.

- *termsFromNts* — a pre-calculated dictionary which has all non-terminals as key and as a value minimum amount of terminals that can be generated from this non-terminal. This is used in the pruning heuristic TL.

- timeLimit — after how long should the tree search or WK-CYK time out

A constructor of the *cWK_CFG* class requires as parameters a list of non-terminals, a list of terminals, the starting non-terminal, the list of rules and the list of relations. An example of an object construction would then be (for rules definition, see 5.2):

```
g = cWK_CFG(['S', 'A'], ['a', 'b'], 'S', rules, [('a', 'a'), ('b', 'b')])
```

The class *cWK_CFG* has these key functionalities:

- initialization, backup and restore

- the run of the tree search algorithm

- pruning heuristics

- node precedence heuristics

- the transformation of grammar to the WK-CNF

- the run of the WK-CYK algorithm

5.1.1 Initialization, backup and restore

During an initialization of the class, several methods are run ensuring validity of the grammar and pre-calculating data.

- method *is_consistent* verifies that the constructor parameters that define the grammar are consistent: sets of terminals and non-terminals must be exclusive, the start symbols and all rules left-hand sides must be found among the non-terminals, rule right hand sides must contain only specified terminals and non-terminals and the complementarity relation list must refer only to the defined terminals. If the method fails, an exception is thrown and the class is not created.
- method *generate_rule_dict* parses the set of rules and creates *ruleDict*
- method *generate_relation_dict* parses the complementarity relation and creates *relDict*
- method *find_erasable_nts* creates the set *erasableNts*, a set of non-terminals that can be erased by applying a sequence of rules. The set is empty if the grammar contains no λ-rule.
- method calc_nt_distances creates the dictionary *ntDistances* mapping non-terminals to the minimum number of rules needed to reach a terminal word
- method calc_min_terms_from_nt creates the dictionary *termsFromNts* mapping non-terminals to the minimum number of terminals that they can generate
- method calc_rules_nt_lens calculates the rule length for all rules

A rule length is a value which indicates how the minimal length of a word will be changed after the application of the rule. It is used by the TL (total length) pruning heuristic. The rule length is equal to the negative left-hand side non-terminal length plus lengths of all elements on the right-hand side.

For instance, let us have a word $A\binom{ab}{c}B$, the rule being applied $A \to B\binom{d}{d}$ and the minimum number of terminals generated from A is 2 and from B is 3. The resulting word will be $B\binom{dab}{dc}B$. The word at the start had total length of 8 (1 for each terminal symbol, 2 for A, 3 for B). The word afterwards has a total length of 11 (3 for each B and 1 for each terminal). Therefore, the rule $A \to B\binom{d}{d}$ must have the length of 3. And it does: -2 for the non-terminal A on the left-hand side, 1 for each of the two terminals and 3 for the non-terminal B.

Since the grammar is able to apply certain transformations (like to the Wk-CNF or removing lambda rules), it is convenient to be able to save the state of the grammar and later restore it. This is what the methods *backup* and *restore* are for. The *backup* method simply saves a copy of the sets of rules, non-terminals and terminals as *rulesBackup*, *ntsBackup* and *tsBackup*. The *restore* method then restores these backup sets and runs again the pre-calculating methods similarly to the class initialization phase.

5.1.2 Run of the state space search algorithm

The state space search algorithm is run by the *run_tree_search* method which has one parameter: the string to be tested for the language membership. It uses a priority queue (python *PriorityQueue* class from the *queue* module) to store the nodes of the search tree. When getting items from this queue, it returns the smallest value item that it contains. The items in the queue are objects of the class *cTreeNode* (described in 5.2). The method also uses a set of hash codes (acquired by the standard Python *hash* function) of all the nodes that have been put in the queue (whether they are still there or have been taken out), so that duplicate nodes are skipped.

At the beginning, the queue contains one node — the starting non-terminal. If the queue is empty (and there is no node being parsed), it indicates that the whole state space has been searched and the method ends with a negative response. Otherwise, it gets the next node from the queue and generates all possible successors of this node (method *get_all_successors*). Each of the successor nodes is tested whether it is the desired solution (method *is_result*). If so, the search optionally prints the path to the solution and ends with the positive result. Otherwise, it checks whether the node is new (has not been in the queue before) by computing its hash and looking into the set of hashes of all generated states. If it is new, it is added to the queue. Also, the main loop checks during every iteration whether the time limit has been exceeded. If so, it returns an empty response (*None* value).

The return value of the method is a tuple containing following items: a maximum number of items in the queue, number of all generated nodes, list of pruning statistics, the actual result (*True*, *False*, or *None*).

The *get_all_successors* method requires two parameters — a node and the input string (which is then passed to the pruning methods). First, it finds the first non-terminal in the given word. It applies all the rules (method *apply_rule*) that the grammar has for this non-terminal, each time creating a new node. The node is then checked by all the pruning algorithms (method *is_word_feasible* described in next section) to see, if the node can lead to the solution. If so, the priority of the node is calculated (method *calculate_distance* described in the next section) and the node is yielded as a result.

The *is_result* method needs to check all the conditions that the node has to meet in order to be a solution (word contains only one letter, the letter is a segment of terminals, its upper strand and lower strand have the same length, all the symbols from the two strands correspond to the complementarity relation and the upper strand is equal to the input string). It takes a word and the input string as parameters and returns a boolean value indicating whether the word is the solution.

The *apply_rule* method takes three parameters: a word, an index of the non-terminal to be replaced and a rule right-hand side. It removes the non-terminal specified by its index (because there can be more than one occurrence of this non-terminal in the word) from the word, and replaces it with the word snippet specified by the rule left-hand side. It contains logic for merging letters containing terminals if they appear in the word next to each other. It returns the final word.

5.1.3 Pruning heuristics

In order to be able to work flexibly with the pruning methods, the class contains a dictionary (called *pruningOptions*) of the implemented pruning functions and indication whether they are active or not. The method *is_word_feasible* iterates through all items in this dictionary and if the value is *True*, indicating the active pruning, it runs the corresponding method. The pruning methods are:

- *prune_check_strands_len* (SL): checks that the sum of the symbols in the upper and the lower strand is not greater than the input length
- *prune_check_total_len* (TL): checks that the total length of the word (count of all terminal symbols plus lengths of all non-terminals) is not greater then the doubled input length
- *prune_check_word_start* (WS): checks that the starting terminals of the word correspond to the input string
- *prune_check_relation* (RL): checks that the starting terminals meet the complementarity relation constraint
- *prune_check_regex* (RE): checks that the input matches the regular expression based on the word

All the pruning methods require a word and the input string as parameters. They return a boolean value indicating whether the word is feasible or not.

The pruning heuristics can be activated or deactivated by the method *activate*, parameters are the name of the heuristic and a boolean value indicating if it should be active or not. For example:

```
g.activate('RE', True)  # g is an object of class cWK_CFG
g.activate('RL', False)
```

5.1.4 Node precedence heuristics

Just like with pruning, the node precedence functionality needs to be flexible — it must be possible to switch between various node precedence methods. The functionality is implemented by the method *compute_precedence*. This methods simply uses *nodePrecedenceList* and *currentNodePrecedence* to call the right specific method. There are 12 of these methods called *compute_precedence_[name]* where *name* is the name of the specific heuristic (one of NTA, WNTA, TM1, TM2, TM3, NTA_TM1, NTA_TM2, NTA_TM3, WNTA_TM1, WNTA_TM2, WNTA_TM3, NONE). All of these methods take a word and an input string as parameters, they return an integer evaluation of the node.

The node precedence heuristics can be activated by the method *activate*, the name of the chosen heuristic is the only parameter. For example:

```
g.activate('NTA')  # g is an object of class cWK_CFG
```

5.1.5 The transformation of a grammar to the WK Chomsky normal form

The transformation of a WK grammar to WK-CNF is quite similar to the transformation of a standard context-free grammar to the CNF. It is performed by the method *to_wk_cnf* and it takes the following steps:

1. Removing λ-rules — this is done by the method *remove_lambda_rules*
2. Removing unit rules — unit rules are rules in the form of $A \to B$ where A and B are non-terminals. These is performed by the method *remove_unit_rules*.

3. Removing unterminatable symbols — unterminatable symbols are non-terminals, which cannot be transformed into terminal strings by any sequence of rules. It is possible to remove them without affecting the grammar language because if such a rule ever appeared in a word, the whole word would be automatically useless. Any rules containing such a symbol are useless, as well, and are also removed. This is performed by the method *remove_unterminatable_symbols*.

4. Removing unreachable symbols — unreachable symbols are symbols that can never appear in a word because there is no sequence of rules leading from the starting symbol that would be able to generate them. Therefore they can be removed without affecting the grammar language. Any rules containing such a symbol are useless, as well, and are also removed. This is done by the method *remove_unreachable_symbols*.

5. Dismantling rules generating terminal letters — WK-CNF requires all rules generating terminals to generate one symbol only, this means only rules in the form of $A \rightarrow \binom{a}{\lambda}$ and $A \rightarrow \binom{\lambda}{a}$ are allowed. This is achieved by the method *dismantle_term_letters* which iterates through all the rules and replaces any terminal letters at the rule right-hand side with a newly generated non-terminal. Afterwards, new rules are added and non-terminals are created which assure that the terminal letter is generated in steps, each rule having at maximum two items on the right-hand side.

   If the rule is $A \rightarrow AB\binom{ab}{c}$ (A, B being non-terminals, a, b, c being terminals), then the A and B are skipped and the letter $\binom{ab}{c}$ is replaced by a new non-terminal N_1. Then, another rule is created: $N_1 \rightarrow \binom{ab}{c}$, which needs to be broken down further. For each terminal in this letter, a new rule is created and the terminal replaced by a new non-terminal, until there remains only one terminal in that letter. The final set of rules is then going to be: $A \rightarrow ABN_1$, $N_1 \rightarrow \binom{a}{\lambda}N_2$, $N_2 \rightarrow \binom{\lambda}{c}N_3$, $N_3 \rightarrow \binom{b}{\lambda}$

6. Dismantling rules generating non-terminals — in the final stage of the transformation, the method *transform_to_wk_cnf_form* iterates through all the rules, it keeps the rules in the WK-CNF form ($A \rightarrow BC$, $A \rightarrow \binom{a}{\lambda}$ and $A \rightarrow \binom{\lambda}{a}$) and breaks down other rules in a process analogous to the actions of *dismantle_term_letters*.

7. Recalculation of data — runs the methods that pre-calculate data, similarly to the class initialization phase or after *restore* method

The methods which remove λ-rules, unit rules, unterminatable symbols and unreachable symbols are useful even outside of the transformation to the WK-CNF. Removing unterminatable symbols and unreachable symbols (and rules containing these symbols) are optional but useful steps in the transformation. The dismantling of rules are two steps that make sense only in this context.

5.1.6 Run of the WK-CYK algorithm

The WK-CYK algorithm is implemented by the method *run_wk_cyk* which takes as a single parameter an input string and returns a boolean value indicating whether the input has been accepted or not. Similarly to tree search, every iteration of the algorithm's main loop checks the elapsed time and if it exceeds the time limit, it returns an empty value (*None*). The implementation follows closely the description in [11] (in section 6) and 3.1. The *run_wk_cyk* method corresponds to the *sets construction* procedure. The *compute set* procedure called from *sets construction* then corresponds to the *computeSet* method of the cWK_CNF class.

5.2 Implementation of the rule and node classes

A rule of a grammar is modeled by the class *cRule*. It contains the following data:

- *lhs* — a non-terminal, left-hand side of the rule
- *rhs* — a word, right-hand side of the rule
- *ntsLen* — length of all non-terminals of the right-hand side
- *upperCnt* — count of all terminals in the upper strand of the right-hand side
- *lowerCnt* — count of all terminals in the lower strand of the right-hand side

The items *ntsLen*, *upperCnt* and *lowerCnt* are there for optimization purposes. It is always possible to iterate over the word and count them, but it is more efficient to count it once for every rule and store this value.

The *cRule* class then contains the following methods:

- *compactize* — this method is called during the object initialization phase and ensures that the right-hand side does not contain any terminal letters next to each other, if it does, then these are merged together. For instance a rule $A \rightarrow \binom{a}{b}\binom{a}{b}$ is transformed to an equivalent rule $A \rightarrow \binom{aa}{bb}$.
- *calculate_cnts* — method is called during the object initialization phase and counts values for *upperCnt* and *lowerCnt*. The length of non-terminals is calculated during the initialization of the cWK_CFG object because it needs to know the length of non-terminals.

A constructor of the *cRule* class requires a left-hand side of the rule, which is a non-terminal, and a right-hand side of the rule, which is a word, i.e. a list of letters. A letter is either a non-terminal or a tuple of two lists. An example of a rule object creation for rule $A \rightarrow A\binom{ab}{\lambda}$ would then be:

```
cRule('A', ['A', (['a', 'b'], [])])
```

A tree node is modeled by the class *cTreeNode* which contains the following:

- *word* — the actual word of the grammar
- *upperStrLen* — number of upper strand terminals in the word
- *lowerStrLen* — number of lower strand terminals in the word
- *ntLen* — length of all non-terminals
- *parent* — node in the search tree, which is this node's predecessor, this is not necessary for the search but once a solution is found, it may be useful to know what path has been taken to reach it
- *hashNo* — a unique hash code of the node calculated during the object initialization

- *precedence* — a value assigned by the active node precedence heuristic, it is used to compare two objects of this class which is needed by the priority queue to order the nodes

A *cTreeNode* constructor requires a word, three integers specifying the *upperStrLen*, *lowerStrLen* and *ntLen*, the parent node, and the precedence. Of course, the lengths of terminals could be counted by the tree node itself during the initialization or when needed. These values are passed to the constructor for optimization purposes. Counting them would require iterating over the whole word which can be quite long. When creating a new node, it is more efficient to take these counts from the parent node and add or subtract differences which are stored in the rule object which is being applied.

An example of a creation of this class object (in this case the root node) could then be:

```
cTreeNode(['S'], 0, 0, 1, None, 1)
```

Both of these classes, *cTreeNode* and *cRule*, as well as the main class cWK_CFG, are in the source file *lib/ctf_WK_grammar.py*, as they are quite closely related.

5.3 Implementation of the test runner class, test scripts and grammars

The grammars that are used for testing both the tree search and the WK-CYK are stored in the file *lib/grammars.py*. Each grammar is characterized primarily by its rules, those are created first. Then the instance of the cWK_CFG is created and then a generator of inputs is assigned to each grammar.

A generator of inputs is a method of each grammar object called *input_gen_func*. Its purpose is to generate inputs for the given grammar of increasing lengths. It takes 3 parameters: a starting number of characters, a step — how many characters should be added in the next generated input, and a boolean indicating whether the generated inputs should be accepted by the grammar or not. The generator does not have to return the input string exactly of the length which it was specified. Sometimes it is not even possible. The generated string's length may be approximate to the specified values.

Here is an example of input generator use for grammar 1:

```
generator = g1.input_gen_func(5, 2, True)
input1 = next(generator) # generates 'aaaaa'
g1.can_generate(input1)
input2 = next(generator) # generates 'aaaaaaa'
g1.can_generate(input2)
```

I have implemented the following five test scripts, which are in the root directory:

- *ts_node_precedence_tests.py* — runs a test for all of the 40 grammars (20 in the basic form and 20 in the WK-CNF) with inputs which are going to be accepted (node precedence is irrelevant if inputs are eventually rejected). Each test runs the tree on search this input one time for each of the available node precedence heuristics.

- *ts_pruning_tests.py* — runs two tests for each of the 40 grammars, one with an input that will be accepted and one that will be rejected by the search. Each test runs the search with all pruning heuristics inactive, then with all active, and then for each one it runs with activated all but the one heuristic. Thus comparing how each one heuristic contributes to the overall performance.

- *ts_speed_tests.py* — runs two tests (one positive, one negative) for each of the 40 grammars, in each test a Tree search is run repeatedly (up to 30 times or it is stopped if a search exceeds time limit) with increasing input length. This is used to analyze the time and memory complexity of the tree search with respect to the input length.
- *ts_var_inputs_tests.py* — runs tests for some hand-picked inputs of the same length in order to compare, how the different variants of the same length inputs affect the performance
- *wk_cyk_tests.py* — runs two tests (one positive, one negative) for 17 grammars, which are ready for WK-CYK run. Those must be in the WK-CNF and grammars 5, 19 and 20 are excluded, since they use other complementarity relation then identity. In each test, the grammar runs the WK-CYK repeatedly with increasing input lengths.

Each of these scripts prints its output into a table where all the results are compared. Outputs, which I received by running these test scripts, are attached in the output directory.

All these scripts use the *cPerfTester* class, which is a sort of middle layer between the grammar class and the test scripts. It helps with displaying the result table and gathering and parsing data returned by the algorithm runs. It contains the following methods:

- *run_test_ntimes* — runs the tree search several times, calculates and returns averages over results of these runs
- *run_node_precedence_test* — a wrapper used by the script *ts_node_precedence_tests.py*
- *run_prune_test* — a wrapper used by the script *ts_pruning_tests.py*
- *run_speed_test* —a wrapper used by the script *ts_speed_tests.py*
- *var_inputs_test* —a wrapper used by the script *ts_var_inputs_tests.py*
- *run_wk_cyk_test* —a wrapper used by the script *wk_cyk_tests.py*

5.4 How to use the application

It is possible to directly run the one or more of the test scripts from the root folder. They do not have other requirements than having installed the interpreter of language Python (version 3). In the application root directory in Linux terminal it can be run simply by typing:

```
python3 ts_node_precedence_tests.py
python3 ts_pruning_tests.py
...
```

In order to use the *cWK_CFG* class directly there is a demo script in the root directory *demo.py* which shows the use of predefined grammars and creating a new grammar step by step and can be called in the same way:

```
python3 demo.py
```

In a nutshell, when using a predefined grammar, it can be used right after import. To test if a grammar 1 can generate a string *aaaaa*, one could write:

```
from lib.grammars import g1
o, a, p, result = g1.can_generate('aaaaa')
print(result)
```

And to define a grammar from scratch and, for instance, run the WK-CYK algorithm, one can write:

```
rules = [
  cRule('S', ['S', 'S', 'S']), # S -> S S S
  cRule('S', [(['a'], ['a'])]) # S -> a/a
]
g1 = cWK_CFG(['S'], ['a'], 'S', rules, [('a', 'a')])
g1.to_wk_cnf()
result = g1.run_wk_cyk('aaaaa')
print(result)
```

Chapter 6

Testing the state space search and the WK-CYK algorithm

In this chapter I present the set of grammars that have been used to test the algorithms and then the results of the tests. First, there is the test comparing the node precedence heuristics. Since only one can be active at a time, it is appropriate to start with choosing the one which provides the best overall performance. The tests that follow after that will all use this winning node precedence heuristic. Next, I test pruning heuristics to see if all of them contribute to the overall performance or if it is better to turn some off. This way I get the best configuration of the state space search that is available. In some cases, a different configuration would be more efficient but the goal here is to get the best overall performance.

When the best configuration is known, I test the performance of both the state space search algorithm and WK-CYK with various inputs, especially inputs of increasing lengths and compare the results.

All tests were run in the following environment:

- CPU: AMD Ryzen 5 3600 6-Core Processor
- Memory: 32 GB
- Operating system: Linux, Ubuntu 21.10
- Interpreter: Python 3.9.7

6.1 Watson-Crick grammars used for testing

For the testing of the tree search algorithm and the WK-CYK algorithm, I have used the following Watson-Crick grammars. Unless stated otherwise, the set of non-terminals and the set of terminals is defined simply by the symbols that appear in the rules — all the uppercase letters are non-terminals of the grammar and all the lowercase letters and digits are terminals. The starting non-terminal is S and the complementarity relation is identity. With these specifications in mind the grammar can be defined by the rules only.

1.
$$S \to \binom{a}{a} \mid SSS$$

The accepted language is: $a(aa)^*$

2.
$$S \to \binom{a}{a}S \mid \binom{b}{b}S \mid \binom{c}{c}S \mid \binom{abc}{abc}$$

The accepted language is: $(a+b+c)^*abc$

The aim of this example is to test inputs with the decisive part on the very end. This could prove difficult since the tree search expands the non-terminals from left to right.

3.
$$S \to A\binom{abc}{abc}$$
$$A \to A\binom{a}{a} \mid A\binom{b}{b} \mid A\binom{c}{c} \mid \binom{\lambda}{\lambda}$$

The accepted language is: $(a+b+c)^*abc$

The aim of this example is, again, to test inputs with the decisive part on the very end while, at the same time, the rules are left recursive.

4.
$$S \to Q\binom{a}{a} \mid ABCDEFG$$
$$Q \to QQ \mid ABCDEFG$$
$$A \to \binom{a}{a} \mid \binom{\lambda}{\lambda}$$
$$B \to \binom{b}{b} \mid \binom{\lambda}{\lambda}$$
$$C \to \binom{c}{c} \mid \binom{\lambda}{\lambda}$$
$$D \to \binom{d}{d} \mid \binom{\lambda}{\lambda}$$
$$E \to \binom{e}{e} \mid \binom{\lambda}{\lambda}$$
$$F \to \binom{f}{f} \mid \binom{\lambda}{\lambda}$$
$$G \to \binom{g}{g} \mid \binom{\lambda}{\lambda}$$

The accepted language is: $a?b?c?d?e?f?g? + (a?b?c?d?e?f?g?)^*a$ ($x?$ denotes that the symbol x is optional, i.e. $(x+\lambda)$)

The problematic feature of this grammar may be the fact, that during the transformation of this grammar to the WK-CNF (more specifically, when removing the λ-rules) the number of rules increases rapidly.

5.
$$S \to \binom{a}{t}S \mid \binom{t}{a}S \mid \binom{g}{c}S \mid \binom{c}{g}A$$
$$A \to \binom{c}{g}A \mid \binom{a}{t}S \mid \binom{g}{c}S \mid \binom{t}{a}B$$
$$B \to \binom{c}{g}A \mid \binom{a}{t}S \mid \binom{t}{a}S \mid \binom{g}{c}C$$
$$C \to \binom{a}{t}C \mid \binom{t}{a}C \mid \binom{g}{c}C \mid \binom{c}{g}C \mid \binom{\lambda}{\lambda}$$

The terminals in this grammar refer to the actual nucleobases in the DNA and the complementarity relation mirrors the relations among them: $\rho = \{(a,t),(t,a),(c,g),(g,c)\}$

The accepted language is: $(\{a,t,c,g\}^*ctg\{a,t,c,g\}^*)^*$

This grammar is taken from [10] and is a first step towards an actual analysis of the DNA. In this case, it simply looks for the substring *ctg*.

6.
$$\begin{aligned} S &\to \binom{a}{\lambda}S \mid \binom{a}{\lambda}A \\ A &\to \binom{b}{a}A \mid \binom{b}{a}B \\ B &\to \binom{\lambda}{b}B \mid \binom{\lambda}{b} \end{aligned}$$

The accepted language is: $a^n b^n$ where $n \geq 1$ (symbol x^n denotes n occurrences of the symbol x)

The grammar is taken from [14].

7.
$$S \to \binom{a}{a}S\binom{a}{a} \mid \binom{b}{b}S\binom{b}{b} \mid \binom{c}{c}$$

The accepted language is: wcw^R where $w \in \{a,b\}^*(w^R$ is the reversal of the string $w)$

8.
$$S \to \binom{a}{a}S\binom{a}{a} \mid \binom{b}{b}S\binom{b}{b} \mid \binom{\lambda}{\lambda}$$

The accepted language is: ww^R where $w \in \{a,b\}^*$

9.
$$\begin{aligned} S &\to BL \mid RB \\ L &\to BL \mid A \\ R &\to RB \mid A \\ A &\to BAB \mid \binom{2}{2} \\ B &\to \binom{0}{0} \mid \binom{1}{1} \end{aligned}$$

The accepted language is: $x2y : x,y \in \{0,1\}^* \wedge |x| \neq |y|$

The grammar is taken from [16].

10.
$$\begin{aligned} S &\to T \mid T\binom{p}{p}S \\ T &\to F \mid FT \\ F &\to \binom{e}{e} \mid W \mid \binom{o}{o}T\binom{p}{p}S\binom{c}{c} \mid X\binom{s}{s} \mid \binom{o}{o}Y\binom{c}{c}\binom{s}{s} \\ X &\to \binom{e}{e} \mid \binom{l}{l} \mid \binom{0}{0} \mid \binom{1}{1} \\ Y &\to T\binom{p}{p}S \mid F\binom{d}{d}T \mid X\binom{s}{s} \mid \binom{o}{o}Y\binom{c}{c}\binom{s}{s} \mid ZZ \\ W &\to \binom{l}{l} \mid Z \\ Z &\to \binom{0}{0} \mid \binom{1}{1} \mid ZZ \end{aligned}$$

The accepted language includes regular expressions over symbols 0 and 1 with parentheses (o for opening and c for closing parenthesis), operators $+$ (p), $*$ (s), $\cdot$ (d) and symbols $\emptyset$ (e), ε (l)

The grammar is taken from [6].

11.
$$\begin{gathered}S \to A \mid B \mid AB \mid BA\\ A \to \binom{a}{a} \mid \binom{a}{a}A\binom{a}{a} \mid \binom{a}{a}A\binom{b}{b} \mid \binom{b}{b}A\binom{b}{b} \mid \binom{b}{b}A\binom{a}{a}\\ B \to \binom{b}{b} \mid \binom{a}{a}B\binom{a}{a} \mid \binom{a}{a}B\binom{b}{b} \mid \binom{b}{b}B\binom{b}{b} \mid \binom{b}{b}B\binom{a}{a}\end{gathered}$$

The accepted language is: $\{a,b\}^* \setminus ww$ where $w \in \{a,b\}^*$ — i.e. the complement of the copy language.

12.
$$\begin{gathered}S \to \binom{r}{\lambda}S \mid \binom{r}{\lambda}A\\ A \to \binom{d}{r}A \mid \binom{d}{r}B\\ B \to \binom{u}{d}B \mid \binom{u}{d}C\\ C \to \binom{r}{u}C \mid \binom{r}{u}D\\ D \to \binom{\lambda}{r}D \mid \binom{\lambda}{r}\end{gathered}$$

The accepted language is: $r^n d^n u^n r^n$ where $n \geq 1$

The grammar is taken from [14].

13.
$$\begin{gathered}S \to \binom{a}{\lambda}S\binom{b}{\lambda} \mid \binom{a}{\lambda}A\binom{b}{\lambda}\\ A \to \binom{c}{a}A \mid \binom{\lambda}{c}B\binom{\lambda}{b}\\ B \to \binom{\lambda}{c}B\binom{\lambda}{b} \mid \binom{\lambda}{\lambda}\end{gathered}$$

The accepted language is: $a^n c^n b^n$ where $n \geq 1$

The grammar is taken from [10].

14.
$$\begin{gathered}S \to \binom{a}{\lambda}S \mid \binom{a}{\lambda}A\\ A \to \binom{b}{\lambda}A \mid \binom{b}{\lambda}B\\ B \to \binom{c}{a}B \mid \binom{c}{a}C\\ C \to \binom{d}{b}C \mid \binom{d}{b}D\\ D \to \binom{\lambda}{c}D \mid \binom{\lambda}{d}D \mid \binom{\lambda}{\lambda}\end{gathered}$$

The accepted language is: $a^n b^m c^n d^m$ where $n, m \geq 1$

The grammar is taken from [10].

15.
$$\begin{gathered}S \to \binom{a}{\lambda}S \mid \binom{b}{\lambda}S \mid \binom{c}{\lambda}A\\ A \to \binom{a}{a}A \mid \binom{b}{b}A \mid \binom{\lambda}{c}B\\ B \to \binom{\lambda}{a}B \mid \binom{\lambda}{b}B \mid \binom{\lambda}{\lambda}\end{gathered}$$

The accepted language is: wcw where $w \in \{a,b\}^*$

The grammar is taken from [10].

16.
$$S \to \binom{a}{\lambda}S\binom{a}{a} \mid \binom{a}{\lambda}A\binom{a}{a}$$
$$A \to \binom{bb}{a}A \mid \binom{bbb}{a}A \mid \binom{\lambda}{b}B$$
$$B \to \binom{\lambda}{b}B \mid \binom{\lambda}{\lambda}$$

The accepted language is: $a^n b^m a^n$ where $2n \leq m \leq 3n$

The grammar is taken from [10].

17.
$$S \to SS \mid \binom{a}{a}S\binom{b}{b} \mid \binom{a}{\lambda}S \mid \binom{a}{\lambda}A$$
$$A \to \binom{b}{a}A \mid \binom{b}{a}B \mid \binom{b}{a}$$
$$B \to \binom{\lambda}{b}B \mid \binom{\lambda}{b} \mid BB \mid \binom{a}{a}S\binom{b}{b} \mid \binom{a}{\lambda}S \mid \binom{a}{\lambda}A$$

The accepted language is: $w : \#_a(w) = \#_b(w)$ and for any prefix v of $w : \#_a(v) \geq \#_b(v)$ where $\#_a(x)$ denotes the number of occurrences of symbol a in the string x

The grammar is taken from [11].

18.
$$S \to \binom{l}{\lambda}S \mid \binom{l}{\lambda}A$$
$$A \to \binom{r}{l}A \mid \binom{r}{l}B$$
$$B \to \binom{l}{r}B \mid \binom{\lambda}{r}B \mid \binom{\lambda}{\lambda} \mid A$$

The accepted language is: $(l^n r^n)^k$ where n does not increase for subsequent k. For instance: *lllrrrlrlr* is within the language, *llrrlllrrr* is not.

The grammar is taken from [10] (where it is stated that the language of this grammar is $(l^n r^n)^k$ for $n, k \geq 1$ which is not correct). The original symbols for opening and closing parenthesis have been replaced by letters l (left parenthesis) and r (right parenthesis).

19. The grammar is identical to the grammar 13 with a difference in the complementarity relation. The relations between symbols a, b and symbols a, c are added. This means that the relation is: $\rho = \{(a,a),(b,b),(c,c),(a,b),(b,a),(a,c),(c,a)\}$

    The accepted language is: $a^n b^m c^n$ where $n, m \geq 1$

20. The grammar is identical to the grammar 14 with a difference in the complementarity relation. The relation between symbols a, b is added making the relation $\rho = \{(a,a),(b,b),(c,c),(d,d),(a,b),(b,a)\}$

    The accepted language is: $a^m b^n c^o d^p$ where $m, n, o, p \geq 1 \wedge m + n = o + p$

There are twenty grammars altogether. Grammars 1–5 are regular, 6–11 are context-free and 12–18 are context-sensitive. Grammars 19 and 20 are context-free but they also have a non-bijective complementarity relation.

In reality, there are not 20 but 40 grammars, because all of them can be used in the basic form and after the transformation to the WK Chomsky normal form. That results in a different grammar (although accepting the same language) which is usually significantly more difficult to calculate with, as there are more rules and many rules generate more non-terminals.

6.2 Testing the state space search

There is a lot of parameters that could be tested and analyzed. How efficient are the various heuristics (both pruning and node precedence) for different grammars. What input lengths are answered within a reasonable time frame? Or more generally, what is the relation between input length and time to get the answer? What are the memory requirements? What is the difference in decision time between input strings which are in the given language and those which are not? Are there significant differences between some inputs of the same lengths?

In order to analyze these questions, I have decided to test the state space search in the following stages.

1. comparison of the node precedence heuristics and analysis of their efficiency
2. comparison of the pruning heuristics and analysis of their efficiency
3. analysis of the time and memory complexity based on the length of the input string
4. testing if there are any different inputs of the same length which would result in a significant difference in the computation complexity
5. testing the WK-CYK algorithm with various grammars and inputs and comparison to the state space search

6.2.1 Comparison of the node precedence heuristics efficiency

In section 4.3, 12 node precedence heuristics have been described and only one of them can be active at a time. In order to compare their effectiveness I used the script *ts_node_precedence_tests.py* which runs one test for each of the 40 grammars with an input that will be accepted. It is not useful to test the node precedence heuristics with inputs that are not within the given language as in such cases, the whole space state needs to be searched and node precedence cannot help in any way. The input strings have been chosen to have suitable lengths, so that the computation is finished (at least with some heuristics) in a reasonable time, specifically within the time limit of ten seconds but the search also should last some measurable amount of time.

Each of the 40 tests consists of 12 runs, each with a different node precedence heuristic active. There are three metrics to observe:

- How many times the search timed out?
- What is the total time in which all 40 tests were completed for each of the heuristics. There should be a kind of penalty involved if the test times out because the time needed for the computation is, in that case, certainly greater then the time it actually ran before it was stopped by the time limit. Therefore, for the sake of the comparison, the time of the search is in this case doubled.
- The total time normalized for each test — all the times are multiplied by a number $n = 1/t_{min}$ where t_{min} is the time of the fastest heuristic for the given test. This is probably the most telling metric as each test has roughly the same impact on the final number.

Each test prints out the results in a table similar to 6.1. The upper part of the table displays the description of the accepted language, number of rules, non-terminals and terminals in the grammar, string on the input (if it is too long to be printed out, its beginning part and total length is printed out instead), whether the input is expected to be accepted and the time limit. The lower part the table shows for each of the node precedence heuristic how much time it took, how many states at most were in the queue at a time and how many states were analyzed, how many times each pruning heuristic was successful and the result of the search (True, False or Timeout). It would not be practical to present here the complete output but it can be recreated simply by running the script again and is also attached in the file *output/node_precedence_test_output.txt*.

Figure 6.1: An output of a node precedence heuristics test

Test 1				
Grammar	$a(aa)^*$			
Rules / NTs / Ts	2/1/1			
Input string	aaaaaaaaaaaaaaaaaaaaaaaaa... [len 801]			
Should accept	Yes			
Timeout	7 seconds			
Strategy	Time	States Q+C	Prunes (SL, TL, WS, RL, RE)	Accepted
NTA	0.4886	994 + 3001	0, 5, 0, 0, 250	TRUE
WNTA	0.4189	498 + 2503	0, 3, 0, 0, 250	TRUE
TM1	0.739	1489 + 3994	0, 7, 0, 0, 249	TRUE
TM2	0.7385	1489 + 3994	0, 7, 0, 0, 249	TRUE
TM3	0.7366	1489 + 3994	0, 7, 0, 0, 249	TRUE
NTA+TM1	0.5903	992 + 2999	0, 5, 0, 0, 250	TRUE
NTA+TM2	0.5915	992 + 2999	0, 5, 0, 0, 250	TRUE
NTA+TM3	0.5923	992 + 2999	0, 5, 0, 0, 250	TRUE
WNTA+TM1	0.495	498 + 2503	0, 3, 0, 0, 250	TRUE
WNTA+TM2	0.4949	498 + 2503	0, 3, 0, 0, 250	TRUE
WNTA+TM3	0.4977	498 + 2503	0, 3, 0, 0, 250	TRUE
NONE	2.4445	4188 + 22147	0, 89, 0, 0, 239	TRUE

It is interesting to notice how different heuristics are better in different test cases. This is illustrated by selected test cases which are on figure 6.2. There are some cases where the best heuristic is the empty one which assigns zero to each node, like in the case of test 23. This is because this heuristic is the simplest one to compute and if no heuristic is effective in a particular test case, this one wins. But since it does not win by a large margin, these cases do not have a decisive impact on the overall result. There was only one timeout of heuristic TM2 and the empty heuristic. The significant result differences indicate that the state space search can be customized to fit a specific grammar and thus further improve its performance.

In some cases, a certain heuristics do not work so well, but their combination does. This can be seen in test 22 — NTA and WNTA have poor result, comparable to no heuristic. TM1, TM2, TM3 have somewhat better result, but by far the best result is achieved by combination of NTA with any version of TN.

The figure 6.3 shows the total result for all of the 40 tests. The left bar of each heuristic shows the total time for the 40 tests and the right bar shows the normalized time. It turns out that the best results are achieved by the combination NTA+TM1. The best

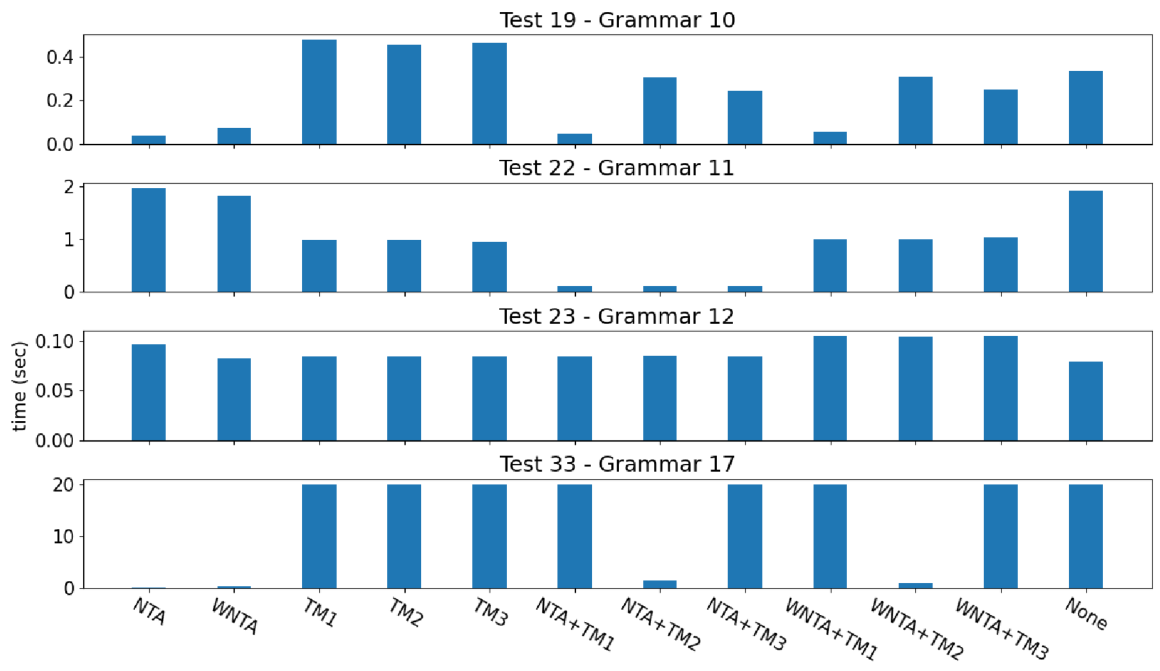

Figure 6.2: Selected comparisons of node precedence heuristics

time overall was achieved by TM3 by very narrow margin. The normalized results of all NTA+TM1, NTA+TM2 and NTA+TM3 heuristics are very close but the narrow winner is NTA+TM1. Even if, in some cases, there are some faster heuristics, it's usually not by much. Interestingly, even though TM2 turns out to have the worst results of them all, often worse than no heuristic, with the combination of NTA the results are among the best. Anyway, for all of the following tests, the winning node precedence heuristic NTA+TM1 will be used.

6.2.2 Comparison of the pruning heuristics efficiency

Pruning has the advantage of being useful whether the input string is going to be accepted or rejected by the tree search. Also, all of the pruning can be active at the same time. Each node can be tested by all available checks to see whether it can be pruned or not.

The testing is performed over 80 tests — each of the 40 grammars is used for a positive test (where the input will be accepted) and a negative test (the input will be rejected). Each test contains seven runs of the tree search algorithm — one where all pruning heuristics are active, one where all are inactive, and one for each heuristic where all are active except the given one.

Similarly to the node precedence heuristics comparison, the metrics that are important are the total time needed to compute the 80 tests and a number of timeouts for each of the seven cases. The main goal here is not to compare the heuristics to each other and find which one is the best — as they can be active at the same time, it does not matter that much. Rather, the goal is to decide whether each of the pruning heuristics improves the performance or if it is better to turn some off. Therefore the comparison between the case where all heuristics are active and the case where a specific heuristic is inactive and the

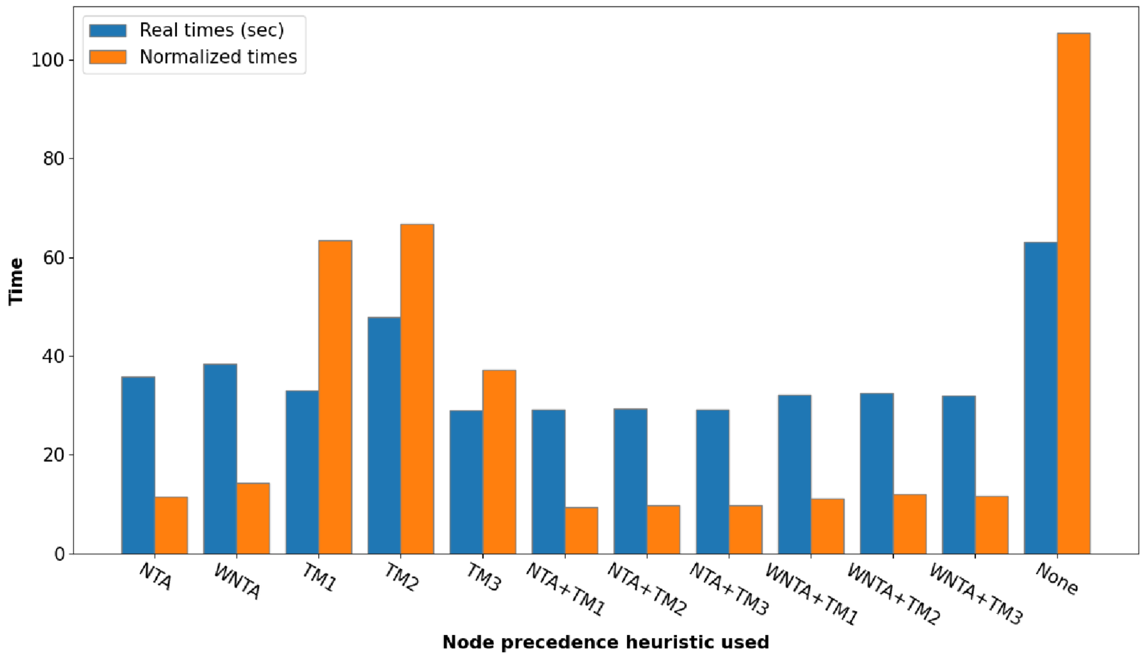

Figure 6.3: Comparison of the node precedence heuristic functions

rest are active is important. If the latter case is faster, the given heuristic's cost (in terms of computing power) is greater than its contribution.

The tests are run by the script *ts_pruning_tests.py* and each test prints out a table similar to the table 6.4. Again, it would not be practical to include the entire output here. The complete set of results can be recreated by running the script again and it is included in the file *output/pruning_test_output.txt*.

Figure 6.4: An output of the pruning heuristics test

Test 1				
Grammar	$a(aa)^*$			
Rules / NTs / Ts	2/1/1			
Input string	aaaaaaaaaaaaaaaaaaaaaaaaa... [len 801]			
Should accept	Yes			
Timeout	7 seconds			
Strategy	Time	States Q+C	Prunes (SL, TL, WS, RL, RE)	Accepted
ALL ON	0.7841	799 + 1200	0, 3, 0, 0, 400	TRUE
strands len OFF	0.7781	799 + 1200	0, 3, 0, 0, 400	TRUE
total len OFF	0.7799	801 + 1200	0, 0, 0, 0, 400	TRUE
terms match OFF	0.7705	799 + 1200	0, 3, 0, 0, 400	TRUE
relation OFF	0.655	799 + 1200	0, 3, 0, 0, 400	TRUE
regex OFF	0.3594	800 + 1599	0, 3, 0, 0, 0	TRUE
ALL OFF	0.2179	801 + 1600	0, 0, 0, 0, 0	TRUE

The summary of results is displayed on figure 6.5 — the number of timeouts and 6.6 — the amount of time for each of the seven cases across the 80 tests. The smaller the individual bars are, the better the result. But in case of the bars representing a specific pruning heuristic being turned off, the bigger the bar, the more important the given heuristic is because the result is that much worse without it.

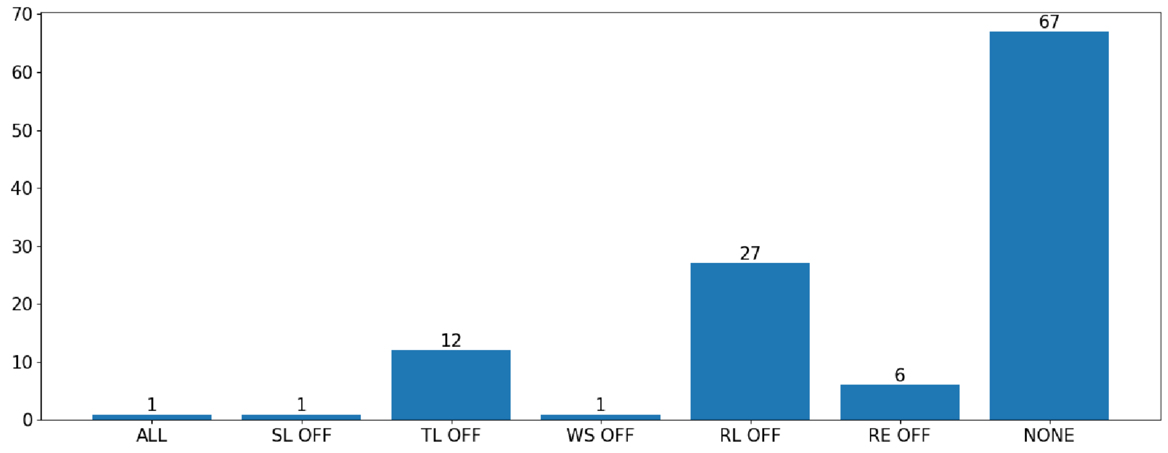

Figure 6.5: The number of timeouts of all pruning tests

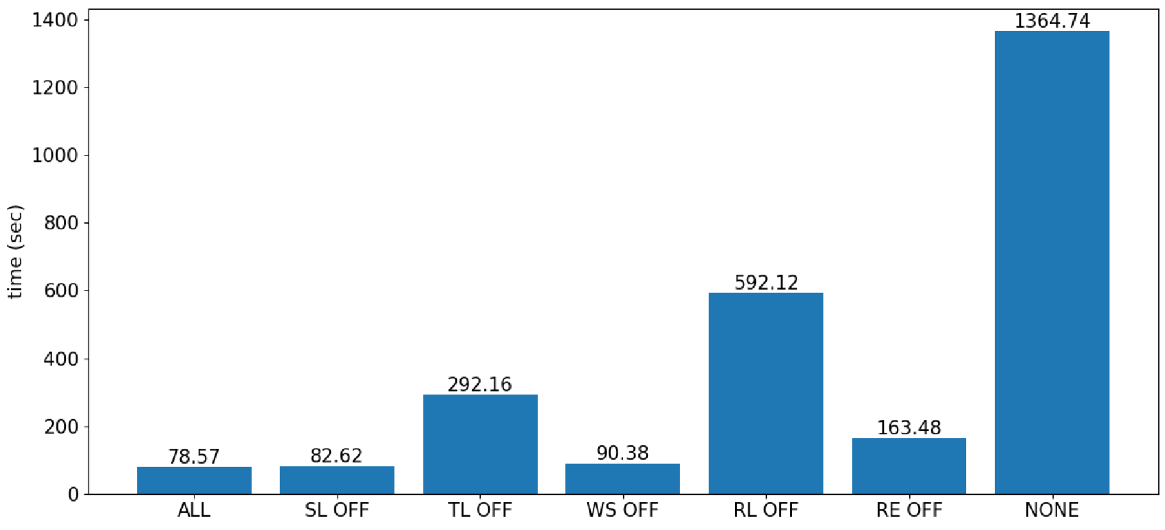

Figure 6.6: The total time of all pruning tests

From these results it is clear that the tree pruning is the key feature of the tree search algorithm. After turning off the pruning, the results are quite poor — 67 out of 80 tests timed out. The total time is then not relevant at all. The middle bars, which represent the individual pruning heuristics being turned off, need to be compared to the first one, where all heuristics are active, to see how important the given heuristic actually is. Thus, the figure suggests, that the RL (complementarity relation) check is the most important one because turning it off had the biggest impact on the result. This can be a bit misleading, as some heuristics can be sometimes backed up by another one. This is the reason why the WS (match of leftmost terminals to the input string) seem to have a rather small impact. If this heuristic is turned off, the dead branch can be identified by the RE (regular expression) check and so the impact is not so big. Similarly, turning off the SL (strands length) heuristic has smaller impact because it is backed up by TL (total length) heuristic.

Nevertheless, all heuristics are useful according to this result because no other result is as good as in the first case where all the heuristics are active. This finding is especially important in the case of the RE heuristic. This one is quite demanding with regards to computational power — regular expression match is performed each time this check is executed. It is the reason why it is the last check that is used, if there is another heuristic able to prune the node, a lot of computational power is saved. However, the figure shows clearly, that the RE heuristic contributes significantly to the overall performance. Still, some tests cannot benefit from all heuristics and turning some off would improve the results. This, again, means that there is some space for improving the performance by customizing the algorithm to specific grammars.

6.2.3 Analysis of the memory requirements

The tree search algorithm needs to keep in memory the nodes which have been generated but not yet analyzed. These are in the priority queue waiting to be used. Also, it keeps track of all the nodes which have already been generated. These are the nodes that had been in the queue before and the nodes that are there at the moment, as in both cases, there is no reason to put them into the queue again. But it is not necessary to keep all these nodes in the memory in order to identify them, their hash code is enough.

If the algorithm should, not only find a solution if there is one, but also find the path from the initial node to the solution, it is not enough just to remember the hashes of nodes that have been analyzed. It is necessary to remember all of the nodes. This is actually the case in the current implementation because when testing the tree search algorithm, it turns out that the memory consumption is not a real issue and it is sometimes convenient to learn the path to the solution. A very simple modification would change this — simply removing the item *parent* from the tree node (*cTreeNode* class) and the method printPath from the grammar (*cWK_CFG* class).

The number of nodes in the queue can go up or down as the search progresses but it is more likely to go up, unless the search is coming to its end. In any case, the important figure is the maximum number of nodes that were in the queue at one point.

Another parameter that needs to be considered is the size of one node in the memory. As mentioned in the section 5, one node contains six integers (storing the number of terminals in the two strands, number of non-terminals, hash number, the node parent and the node precedence evaluation) and the word itself. The word can contain up to twice the number of symbols then is the length of the input string. If it contains more, it is going to be pruned.

This is not necessarily true, if the grammar contains some λ-rules and non-terminals that can be erased. Then the theoretical length of the word in memory has no limit but this is not a typical scenario and it can be avoided altogether by applying the λ-rules removal algorithm.

The equation for getting the memory requirement, based on the number of nodes working with, is than the following:

$$S_{all} \times size(int) + S_{all} \times (6 \times size(int) + 2 \times size(symbol) \times |input|)$$

After the removal of the parent information from the nodes, the equation would be as follows:

$$S_{all} \times size(int) + S_{open} \times (6 \times size(int) + 2 \times size(symbol) \times |input|)$$

S_{all} is the number of all generated nodes, S_{open} is the maximum number of nodes in the queue, $size(int)$ is the size of an integer , $size(symbol)$ is the size of one symbol of the grammar and $|input|$ is the length of the input string.

I've chosen grammar 3 in WK-CNF to test the results in practice, as this grammar is among the hardest ones with respect to the computing complexity. I used inputs in the form of $a^n b$ with increasing n, which will always be rejected, because the grammar accepts strings that end with symbols abc.

The figure 6.7 on the left shows the amount of memory needed in relation to the input length. The middle graph shows the memory consumption in relation to the time needed for computation. To the right is the experimental result where I measured real memory consumption of the program in time.

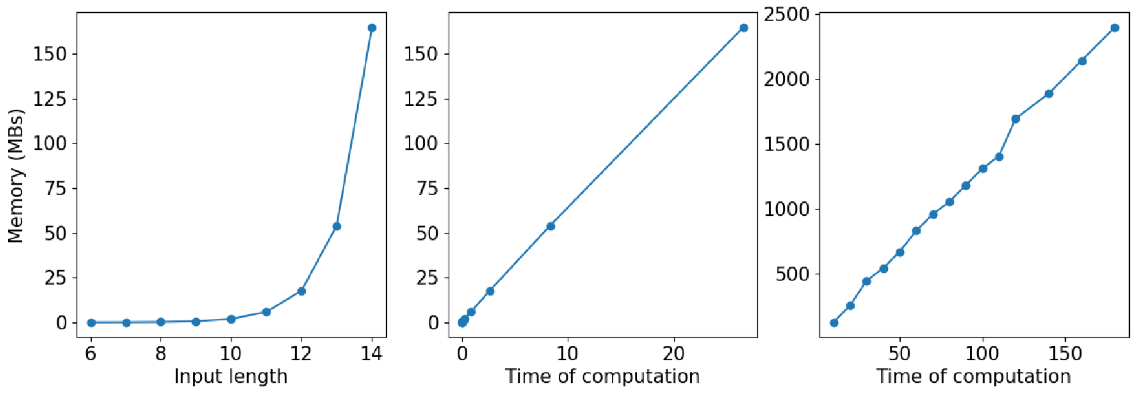

Figure 6.7: Memory consumption of the tree search (the left and middle parts show computed values, the part on the right shows real measurements)

The measurement of the real memory consumption shows that there is a possibility to be limited by the memory available. As the figure 6.7 on the right shows, the tree search used 1 GB of memory after approximately 80 seconds of running and the consumption increases linearly (the number depends on the hardware and environment so it is very crude). On the other hand, it does not seem practical to have the time limit of the computation too high. If the result is not found in the order of tens of seconds or, at most, minutes, it is probably not going to be found (within a reasonable time frame) at all.

6.2.4 The time complexity of the state space search

The previous sections showed that the best overall performance is achieved when using all of the tree pruning heuristics and using the NTA+TM1 as the node precedence heuristic. This may not be the case for every grammar or every input, but it is the case overall. Therefore, this will be the setting used in the sections that follow — testing the tree search performance, analyzing the practical complexity and comparison to the WK-CYK algorithm.

I used the script *ts_speed_tests.py* to test the time complexity with respect to the input length. It runs 80 test, two for each of the 40 grammars — one with inputs that are going to be accepted by the tree search and one with inputs which will be refused. Each test runs the tree search several times and increases the input length. It stops when the computation takes longer than a limit of ten seconds or after 30 runs. For each of the 80 tests, it prints out a table similar to 6.8. As it would not be practical to present all of the results here, they can be recreated by simply running the script again and they are included in the file *output/speed_test_output.txt*.

Figure 6.8: An output of the time complexity test

Test 1				
Grammar	$a(aa)^*$			
Rules / NTs / Ts	2/1/1			
Should accept	Yes			
Timeout	10 seconds			
Input length	Time	States Q+C	Prunes (SL, TL, WS, RL, RE)	Accepted
2001	4.6528	1999 + 3000	0, 3, 0, 0, 1000	TRUE
2101	5.1277	2099 + 3150	0, 3, 0, 0, 1050	TRUE
2201	5.6251	2199 + 3300	0, 3, 0, 0, 1100	TRUE
2301	6.134	2299 + 3450	0, 3, 0, 0, 1150	TRUE
2401	6.665	2399 + 3600	0, 3, 0, 0, 1200	TRUE
2501	7.3398	2499 + 3750	0, 3, 0, 0, 1250	TRUE
2601	8.057	2599 + 3900	0, 3, 0, 0, 1300	TRUE
2701	8.4195	2699 + 4050	0, 3, 0, 0, 1350	TRUE
2801	9.2167	2799 + 4200	0, 3, 0, 0, 1400	TRUE

The performance for different grammars is quite different. In case of 11 of the 20 grammars (specifically, grammars 2, 5–10, 12-14, 19), the resulting graph is a very regular parabola. Often a bit steeper when transformed to the WK-CNF and often steeper when the inputs are going to be rejected. An example is on figure 6.9. The performance with these grammars allows at least hundreds, in most cases thousands, of symbols on the input.

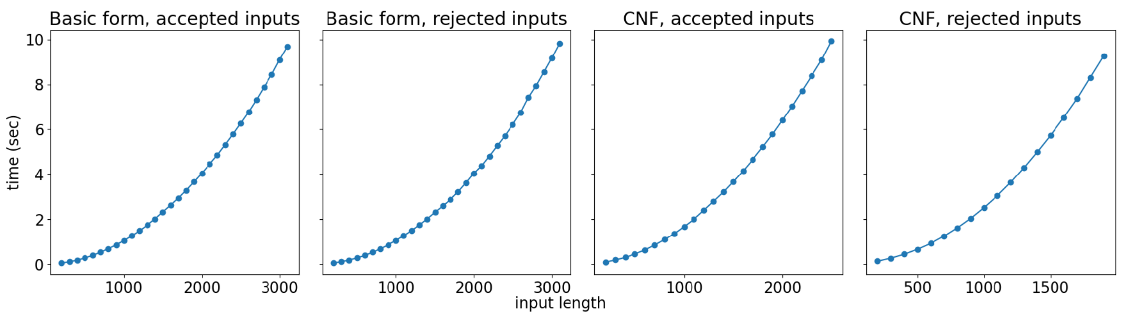

Figure 6.9: Grammar 12: $r^n d^n u^n r^n$

Occasionally, thanks to the pruning heuristics, the algorithm is able to tell practically immediately that there is no solution. This is the case of grammar 3 6.10 and grammar 4 6.11 in basic forms, making the complexity of this particular search constant. The grammar 3 has as the first rule, which it has to use to proceed further, $S \rightarrow A\binom{abc}{abc}$. If the input string does not end with abc, the regular expression check immediately detects that the input cannot be matched, it prunes the only branch and the search is finished.

Similarly, in the case of grammar 4, any inputs that are longer than seven symbols need to end with the symbol a and can be reached only by using $S \rightarrow Q\binom{a}{a}$ as the first rule. The regular expression check immediately prunes this branch. The rest of the tree is searched very quickly because the only other possible starting rule is $S \rightarrow ABCDEFG$, there are not many states that can be reached from it, so this part of the tree is always small.

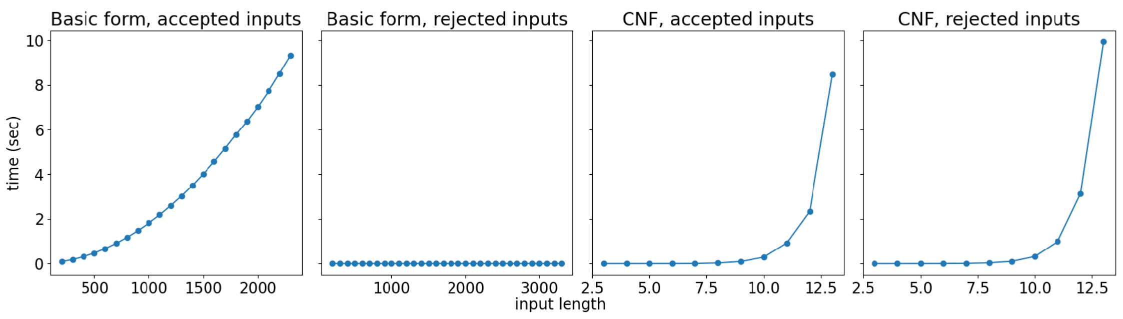

Figure 6.10: Grammar 3: $(a + b + c)^*abc$ with left recursive rules

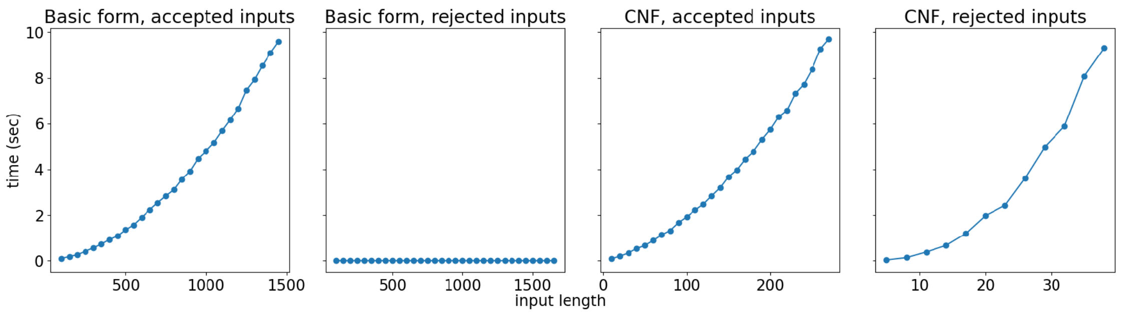

Figure 6.11: Grammar 4: $(a + b + c)^*abc$ with right recursive rules

After conversion of the grammar to the WK-CNF, the complexity usually goes up. The transformation adds a lot more rules and so the state space expands more rapidly, there are also longer paths from the starting non-terminal to the final string (containing only terminals) making the tree deeper. Also, the node precedence heuristics and the pruning have harder time because many rules contain non-terminals only and most of the heuristics work with terminals. The most extreme case is the grammar 3 6.11 where the tree search is very effective for grammar in basic form (as discussed, in case of rejecting inputs the result is immediate) but has very bad effectiveness for this grammar in the WK-CNF. The maximum length of input it can answer within 10 second is about 13–14 symbols.

The worst results for grammar in the basic form are in the case of grammar 17 6.12. Here, the tree search can handle only inputs with length of about 20 symbols within 10 seconds.

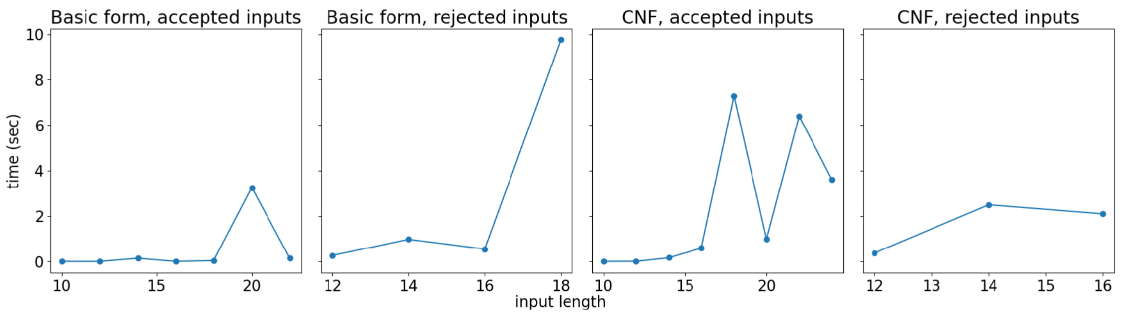

Figure 6.12: Grammar 17: $w : \#_a(w) = \#_b(w)$ and for any prefix v of w: $v : \#_a(v) \geq \#_b(v)$

Some grammars manifest an interesting behavior — for some longer inputs the performance is actually better. This is the case of grammars 11 (6.13 on the left), 16 (6.14 on the middle right) and 18 (6.15 on the middle left and middle right). Some of these results may appear to be partly random, simply the input generator might sometimes generate an input which is more complex and sometimes less complex to compute. This is the case of the grammar 18, here, in fact, the shape of the randomly generated input has a significant impact on the performance which is the reason for the irregular curves. However, repeated tests of the other two grammars confirmed that this happens always and the figure of grammar 16 has a clear pattern, it is certainly not random.

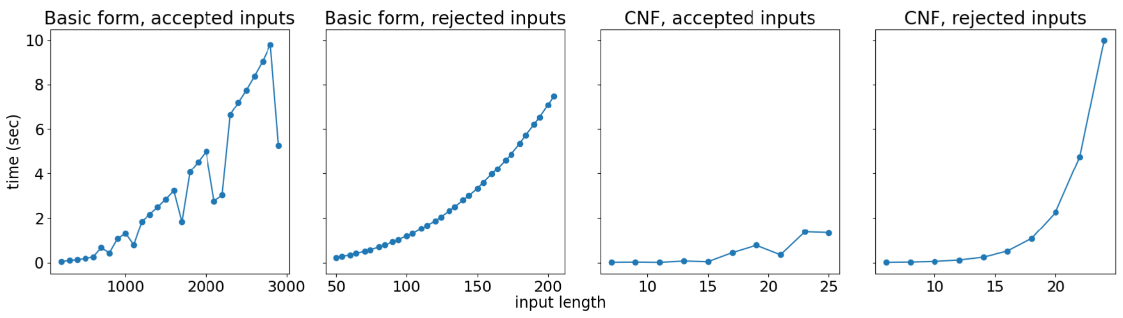

Figure 6.13: Grammar 11: $(ww)^C$

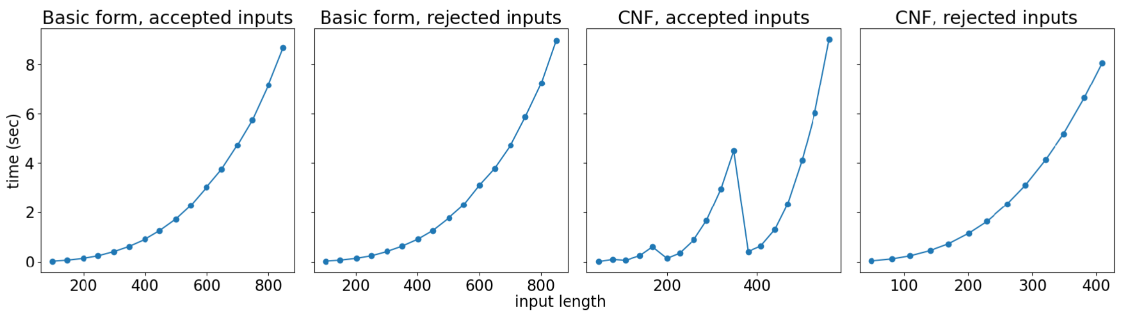

Figure 6.14: Grammar 16: $a^n b^m a^n$ where $2n \leq m \leq 3n$

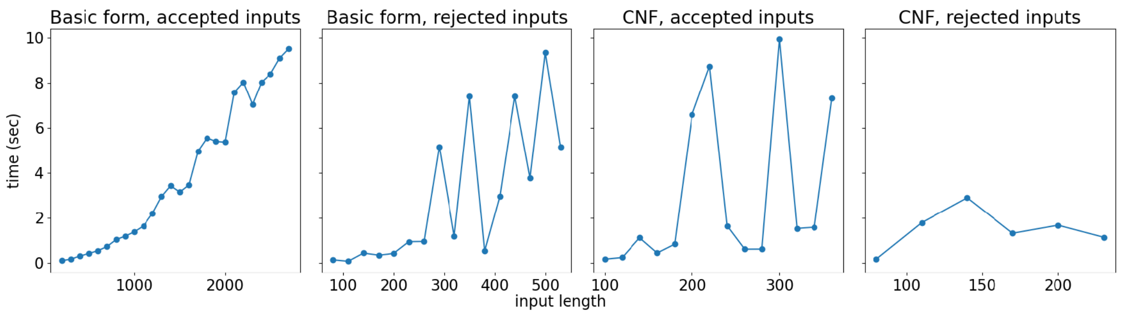

Figure 6.15: Grammar 18: $(l^n r^n)^k$ where n does not increase for subsequent ks

The results of the three grammars which have not been yet mentioned, grammars 1, 15 and 20, are mostly similar to the standard parabola of the majority of tests with some irregularities.

6.2.5 Comparing different inputs of the same length

There are some grammars which do not provide any possibility of an interesting or edge case input. Specifically, it is the case of grammars 1–4. Grammar 1 works only with the terminal symbol a, grammars 2, 3 and 4 are only interested whether a string ends with specific symbols. But the rest of the grammars, grammars 5–20, all provide some space for trying to come up with an edge case input — an input, which has the key part on the very end or on the very beginning etc. I have manually created some of these edge cases and tested what differences in performance there are. For this test I used a script *ts_var_inputs_tests.py*. Its complete output can be again recreated by running the script and is attached in the file *output/var_inputs_test_output.txt*.

I have selected three of these tests to present here. The tables 6.16, 6.17, 6.18 show the output of three tests from the script *ts_var_inputs_tests.py*. The first column called Input displays the string that has been used as an input in a compact format $(ns)^*$ where n is a number of occurrences of the symbol or string s. For instance, $3a$ $2ab$ would translate to string $aaaabab$.

As previously mentioned, one advantage of state space search is that sometimes it can recognize right away that a certain input is not in a given language. It is usually when the string starts with a symbol that cannot be at the beginning. For instance, the second input on table 6.16, where the language is $a^n b^n$ and the input starts with b, can never be accepted. A similar case is the second input in table 6.18 where the language is $r^n d^n u^n r^n$ and it cannot start with anything other than r.

On the other hand, sometimes the search has harder time finding a key part of the string which is at the end. This can take longer as is the case in the last input of 6.16 (search has to get to the end of the string to see that there are not enough b symbols). Some inputs in the table 6.18 also suffer for the same reason. This is usually not a problem that could break the practical use completely. The result is typically still reached within a reasonable time. An extreme example is, however, on the table 6.17 where the last input is not decided in time, even though other inputs were decided very quickly.

Figure 6.16: Test of various inputs for the grammar 6

Test 2				
Grammar	$a^n b^n$			
Rules / NTs / Ts	6/3/2			
Timeout	10 seconds			
Input	Time	States Q+C	Prunes(SL, TL, WS, RL, RE)	Accepted
500a	0.1881	2 + 999	4, 0, 998, 0, 0	FALSE
500b	0.0	1 + 0	0, 0, 2, 0, 0	FALSE
a 500b	0.3448	3 + 502	2, 0, 2, 2, 500	FALSE
500a b	1.0903	3 + 1996	2, 0, 1000, 998, 0	FALSE

Figure 6.17: Test of various inputs for the grammar 8

Test 4				
Grammar Rules / NTs / Ts Timeout	ww^r 3/1/2 10 seconds			
Input	Time	States Q+C	Prunes(SL, TL, WS, RL, RE)	Accepted
2000ab 2000ba a	0.0003	1 + 1	0, 0, 3, 0, 2, 0	FALSE
a 2000ab 2000ba	0.0003	1 + 1	0, 0, 2, 0, 3, 0	FALSE
2000ab a 2000ba	10.0079	1 + 1461	0, 0, 2921, 0, 1, 0	TIMEOUT

Figure 6.18: Test of various inputs for the grammar 12

Test 8				
Grammar Rules / NTs / Ts Timeout	$r^nd^nu^nr^n$ 10/5/3 10 seconds			
Input	Time	States Q+C	Prunes(SL,TL,WS,RL,RE)	Accepted
500r 500d 500u 500r d	8.9568	1500 + 3001	0, 0, 3002, 998, 501, 1	FALSE
d 500r 500d 500u 500r	0.0001	1 + 0	0, 0, 2, 0, 0, 0	FALSE
500r 500d 500u 501r	8.9805	1500 + 3002	2, 0, 3000, 1000, 501, 0	FALSE
500r 500d 501u 500r	3.0577	1000 + 2001	0, 0, 3000, 2, 0, 0	FALSE
500r 501d 500u 500r	1.0785	1000 + 1001	0, 0, 2000, 2, 0, 0	FALSE
501r 500d 500u 500r	1.0763	1001 + 1002	0, 0, 2002, 2, 0, 0	FALSE
r 500d 500u 500r	0.0003	2 + 3	0, 0, 4, 2, 0, 0	FALSE
500r d 500u 500r	0.191	501 + 502	0, 0, 1002, 2, 0, 0	FALSE
500r 500d u 500r	0.9625	1000 + 1003	0, 0, 2002, 2, 0, 0	FALSE
500r 500d 500u r	3.0452	1001 + 2002	2, 0, 3000, 2, 0, 0	FALSE

6.3 Testing the efficiency of WK-CYK

I have tested the WK-CYK algorithm in the similar manner as the tree search. This time not all grammars can be used due to the limitations of WK-CYK. The grammars must be in the WK-CNF and grammars 5, 19 and 20 cannot be used at all, since WK-CYK requires the complementarity relation to be identity which is not the case of these three grammars. Therefore there are 17 grammars that can be tested. The script *wk_cyk_tests.py* runs two tests for each of these grammars. One with inputs that should be accepted and one with inputs that should be rejected. Again, each test increases the input string length until the computation lasts more that the limit of 10 seconds. The output of each test is a table similar to 6.19. The entire output can be recreated by running the script again and is attached in file *output/wk_cyk_test_output.txt.*

It turns out that the WK-CYK gives very similar performance in all tests — for all the grammars and regardless whether the input is accepted or not. The figure 6.20 on the left shows a result for the first test which is very similar to all the others. The limit of ten seconds is reached by WK-CYK when the input has about 33 symbols.

These results confirm the claim made by the authors of WK-CYK that the complexity with regards to the input length is $O(n^6)$. The figure 6.20 on the right shows the same test

Figure 6.19: An output of the WK-CYK time complexity test

Test 1		
Grammar	$a(aa)^*$	
Rules / NTs / Ts	2/1/1	
Should accept	Yes	
Timeout	7 seconds	
Input length	Time	Accepted
3	0.0	TRUE
5	0.0	TRUE
7	0.0	TRUE
9	0.01	TRUE
11	0.03	TRUE
13	0.07	TRUE
15	0.15	TRUE
17	0.29	TRUE
19	0.53	TRUE
21	0.88	TRUE
23	1.45	TRUE
25	2.31	TRUE
27	3.49	TRUE
29	5.23	TRUE
31	7.55	TRUE
33	10.67	TRUE

with the input lengths raised to the power of six which can be considered to be a number of numeric operations needed for the computation. The curve is then very close to linear.

When the results of the WK-CYK and state space search are compared, the advantage of state space search is the actual speed in most cases. The results in the previous sections showed that of all the grammars only one (grammar 17) was slower in the basic form when analyzed by tree search then when analyzed by WK-CYK. After transformation to Wk-CNF two more grammars (grammar 3 and 11) were comparable or slower when analyzed by the tree search. Grammar 1 was slower for the negative inputs.

6.4 Comparison of the state space search and WK-CYK

It has been concluded in the previous section that the WK-CYK algorithm is able to compute within the time limit of ten seconds results for inputs of length of approximately 33 symbols. I assume that this will always be the case even for grammars that would have to be modified in order to be suitable for WK-CYK (grammars 5, 19 and 20). If these results are compared with the results of state space search over grammars in basic forms, only one of them is more efficient with WK-CYK. Other 19 are more efficient with the state space search allowing hundreds of input symbols at minimum. That means that state space search was more efficient in 38 out of 40 test cases (each grammar is tested with accepted and rejected inputs) i.e. in 97.5 % of cases. In this comparison the state space search benefits from being able to work with any WK grammar — there is no need to transform it to the WK-CNF.

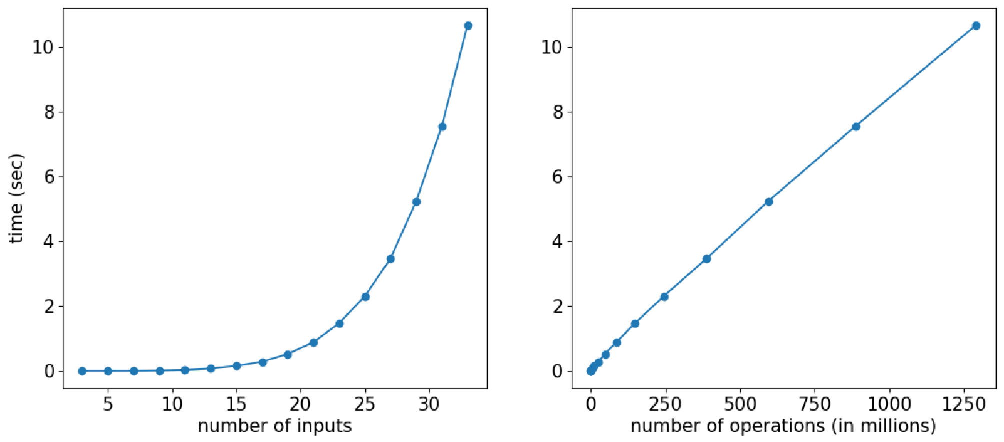

Figure 6.20: WK-CYK test result

If the state space search is compared to WK-CYK over all 40 grammars, WK-CYK has better efficiency in case of grammars 3 and 11 (in the WK-CNF) and with the rejected inputs for grammar 1. That is 9 test cases out of 80 (four test cases of grammar 17, two of grammars 3 and 11, one of grammar 1) i.e. in 88,75 % of cases. However, this comparison assumes that there is a need to use the grammars in the CNF.

These results show some advantages and disadvantages of the two algorithms. An advantage of state space search is the flexibility regarding the grammars. It does not require to work with grammars in the WK-CNF. Also, it does not require the complementarity relation to be identity. Even though it is always possible to transform any WK grammar to the WK-CNF and it is always possible to further transform the grammar in order to use only the identity as the relation, this can significantly add to the grammar's complexity.

A useful feature of the state space search is the fact that it can be configured for the need of a specific grammar. If the membership test will be performed repeatedly on a grammar, it is possible to find out what node precedence heuristic works best and what pruning heuristics are useful in that particular scenario by running tests analogous to those presented in section 6.2.1. Thus the performance may be further enhanced.

An advantage of WK-CYK, on the other hand, is its universality. It has roughly the same speed every time, it does not significantly depend on the grammar (increasing number of rules adds a little bit) and it does not matter, if the input is going to be accepted or not. For very complicated grammars, especially with lots of rules or long derivations from the starting symbol to the final string, WK-CYK still might be more practical.

Chapter 7

Conclusion

In this work I have presented various existing models for representing Watson-Crick languages, most importantly Watson Crick automata and Watson-Crick grammars and I have analyzed the algorithm WK-CYK which is used for testing a membership of strings in languages defined by Watson-Crick context-free grammars. Than I have come up with another algorithm for testing membership in WK languages which I call the state space search or the tree search which is the most important contribution of this thesis.

The state space search is based on a standard Breadth-first search algorithm where the starting non-terminal of the grammar is the root node and every applicable rule creates successors in the tree. The state space search then introduces various optimizations, from which the most important are pruning and node precedence heuristics. Pruning uses five different methods of identifying that a given node cannot produce the desired solution and removes the entire branch. Node precedence heuristics attempt to choose more promising nodes to be analyzed first. I have implemented and tested 12 such heuristics and chosen the one which had the best overall results (called NTA+TM1) as the default one.

I have collected or created twenty Watson-Crick context-free grammars to test the WK-CYK and state space search algorithms. I have implemented both WK-CYK and state space search in the Python language and written scripts to test the algorithms with all these grammars and various inputs. The test results showed that for the majority of the grammars, the state space search was very efficient and it can quickly decide membership problem of inputs that are hundreds or even thousands of symbols long.

Among the advantages of the state space search is the fact that it can work over any forms of WK context-free grammars (not only grammars in the WK-CNF or grammars with the identity complementarity relation). Also, it can be optimized for a particular grammar. Tests comparing performance of node precedence heuristics and tests of pruning heuristics can be run in order to find which combination is best for a particular grammar thus further improving the performance.

Testing the WK-CYK algorithm showed that its theoretical complexity $O(n^6)$ with respect to the input length corresponds to the real performance. In practice, it is able to decide membership problem of inputs up to the length of approximately 30–50 symbols. That is much less than state space search but, on the other hand, its performance is almost identical for any grammar and for any input. It is more efficient in case of some specific or complex grammars where the performance of state space search struggles.

The state space search is a suitable algorithm for parallization. Several processes can take nodes from the queue of open nodes and analyze different branches of the tree independently. This would be a natural next step in the further development of the state space search. It is possible to come up with other heuristics for both the pruning and node precedence. As for pruning, one possibility would be to expand the regular expression matching (RE) heuristic also to consider the lower strand. Another idea for pruning is to calculate how many terminals can be generated at minimum to the lower strand and to the upper strand individually (currently, it is calculated how many terminals a non-terminal produces to both strands) thus making the constraint of the words stronger. As for the node precedence heuristics, it may be worthwhile to use some of the grammars with which state space search is not efficient (in particular grammars 3 in the WK-CNF and grammar 17) and design or improve node precedence heuristics with respect to these particular cases. Then it would be necessary to test all these new heuristics and see if they contribute to the overall performance or not. Another promising improvement could be analyzing the input from both sides at the same time. This could help with the cases, when the key part of the input is at or near its end and the state space search may struggle to get there in a reasonable time frame.

Bibliography

[1] Chatterjee, K. and Ray, K. S. Watson-crick pushdown automata. *Kybernetika.* PRAGUE 8: KYBERNETIKA. 2017, vol. 53, no. 5, p. 868–876. ISSN 0023-5954.

[2] Cocke, J. *Programming Languages and Their Compilers: Preliminary Notes.* USA: New York University, 1969. ISBN B0007F4UOA.

[3] Czeizler, E. and Czeizler, E. Parallel communicating Watson-Crick automata systems. *Acta cybernetica (Szeged).* Szeged: Laszlo Nyul. 2006, vol. 17, no. 4, p. 685–700. ISSN 0324-721X.

[4] Czeizler, E. and Czeizler, E. A Short Survey on Watson-Crick Automata. *Bull. EATCS.* january 2006, vol. 88.

[5] Czeizler, E., Czeizler, E., Kari, L. and Salomaa, K. On the descriptional complexity of Watson–Crick automata. *Theoretical computer science.* Elsevier B.V. 2009, vol. 410, no. 35, p. 3250–3260. ISSN 0304-3975.

[6] Erickson, J. *Context-Free Languages and Grammars* [https://jeffe.cs.illinois.edu/teaching/algorithms/models/05-context-free.pdf]. 2018. Accessed: 2022-10-05.

[7] Freund, R., Paun, G., Rozenberg, G. and Salomaa, A. Watson-Crick finite automata. In: Rubin, H. and Wood, D. H., ed. *DNA Based Computers, Proceedings of a DIMACS Workshop, Philadelphia, Pennsylvania, USA, June 23-25, 1997.* DIMACS/AMS, 1997, vol. 48, p. 297–328. DIMACS Series in Discrete Mathematics and Theoretical Computer Science.

[8] Kasami, T. *An efficient recognition and syntax analysis algorithm for context-free languages.* AFCRL-65-758. Bedford, MA: Air Force Cambridge Research Laboratory, 1965.

[9] Kuske, D. and Weigel, P. The Role of the Complementarity Relation in Watson-Crick Automata and Sticker Systems. In: *Developments in Language Theory.* Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, p. 272–283. Lecture Notes in Computer Science. ISBN 9783540240143.

[10] Mohamad Zulkufli, N. L., Turaev, S., Mohd Tamrin, M. I. and Messikh, A. Generative Power and Closure Properties of Watson-Crick Grammars. *Applied computational intelligence and soft computing.* Hindawi Publishing Corporation. 2016, vol. 2016, p. 1–12. ISSN 1687-9724.

[11] Mohamad Zulkufli, N. L., Turaev, S., Mohd Tamrin, M. I. and Messikh, A. Watson–Crick Context-Free Grammars: Grammar Simplifications and a Parsing Algorithm. *The Computer Journal*. january 2018, vol. 61, no. 9, p. 1361–1373. DOI: 10.1093/comjnl/bxx128. ISSN 0010-4620. Available at: https://doi.org/10.1093/comjnl/bxx128.

[12] Mohamad Zulkufli, N. L., Turaev, S., Tamrin, M. I. M. and Messikh, A. The Computational Power of Watson-Crick Grammars: Revisited. In: *Bio-inspired Computing – Theories and Applications*. Singapore: Springer Singapore, 2017, vol. 681, p. 215–225. Communications in Computer and Information Science. ISBN 9789811036101.

[13] Ray, K. S., Chatterjee, K. and Ganguly, D. State complexity of deterministic Watson–Crick automata and time varying Watson–Crick automata. *Natural computing*. Dordrecht: Springer Netherlands. 2015, vol. 14, no. 4, p. 691–699. ISSN 1567-7818.

[14] Subramanian, K. G., Hemalatha, S. and Venkat, I. On Watson-Crick Automata. In: *Proceedings of the Second International Conference on Computational Science, Engineering and Information Technology*. New York, NY, USA: Association for Computing Machinery, 2012, p. 151–156. CCSEIT '12. DOI: 10.1145/2393216.2393242. ISBN 9781450313100. Available at: https://doi.org/10.1145/2393216.2393242.

[15] Subramanian, K. G., Venkat, I. and Mahalingam, K. Context-Free Systems with a Complementarity Relation. In: *2011 Sixth International Conference on Bio-Inspired Computing: Theories and Applications*. IEEE, 2011, p. 194–198. ISBN 1457710927.

[16] Viola, E. *Context-Free Languages* [https://www.ccs.neu.edu/home/viola/classes/slides/slides-context-free.pdf]. 2016. Accessed: 2022-10-05.

[17] Younger, D. H. Recognition and parsing of context-free languages in time n3. *Information and Control*. 1967, vol. 10, no. 2, p. 189–208. DOI: https://doi.org/10.1016/S0019-9958(67)80007-X. ISSN 0019-9958. Available at: https://www.sciencedirect.com/science/article/pii/S001999586780007X.