ČESKÉ VYSOKÉ UČENÍ TECHNICKÉ V PRAZE

Fakulta jaderná a fyzikálně inženýrská

VÝZKUMNÝ ÚKOL

Praha, 2014 Jakub Klemsa

ČESKÉ VYSOKÉ UČENÍ TECHNICKÉ V PRAZE

Fakulta jaderná a fyzikálně inženýrská

Katedra matematiky

VÝZKUMNÝ ÚKOL

Modely sebeskládajících DNA nanostruktur

Models of self-assembling DNA nanostructures

Vypracoval: Jakub Klemsa
Školitel: Ing. Štěpán Starosta, Ph.D.
Akademický rok: 2013/2014

Na toto místo přijde svázat **zadání mého výzkumného úkolu**!

V jednom z výtisků musí být **originál** zadání, v ostatních kopie.

Čestné prohlášení

Prohlašuji na tomto místě, že jsem předloženou práci vypracoval samostatně a že jsem uvedl veškerou použitou literaturu.

V Praze dne June 9, 2014

. . . .
Jakub Klemsa

Poděkování

Děkuji Ing. Štěpánu Starostovi, Ph.D., za vedení mého výzkumného úkolu a za podnětné návrhy, které ho obohatily.

Jakub Klemsa

Název práce: **Modely sebeskládajících DNA nanostruktur**

Autor: Jakub Klemsa

Obor: Inženýrská informatika

Zaměření: Matematická informatika

Druh práce: Výzkumný úkol

Vedoucí práce: Ing. Štěpán Starosta, Ph.D., FIT ČVUT

Konzultant: —

Abstrakt: DNA výpočty slibují nové možnosti pro řešení těžkých výpočetních problémů. Od prvního výpočetního experimentu s DNA, který provedl Adleman v roce 1994, bylo provedeno mnoho práce v popisu jak matematických tak fyzikálních vlastností těchto systémů. V této práci se zaměřuji na matematickou informatiku, speciálně na výpočetní proveditelnost. Představuji čtyři druhy prostředků, které DNA algoritmy užívají, a dávám vztah ke "klasickým" zdrojům Turingova stroje jako jsou čas a prostor. Poté odvozuji model vhodný pro řešení NP-úplných problémů. Dva NP-úplné problémy a jeden NP problém následně řeším pomocí tohoto modelu, jeden z nich také simuluji s použitím simulátoru DNA skládání.

Klíčová slova: DNA výpočty, třídy složitosti, Turingův stroj, Wangovy dlaždice.

Title: **Models of self-assembling DNA nanostructures**

Author: Jakub Klemsa

Abstract: DNA computing promises new possibilities in solving hard computational problems. Since first computational DNA experiment by Adleman in 1994 much work has been done to describe both mathematical and physical phenomenas. In this work we focus on mathematical informatics, especially on computational feasibility. We present four kinds of resources to be studied in case of DNA algorithms and give a relation to "classical" resources of Turing machine such as time and space.

Then we derive a model suitable for solving NP-complete problems. Two NP-complete problems and one NP problem are then solved using this model, one of them is also simulated using DNA tile assembly simulator.

Keywords: DNA computing, complexity classes, Turing machine, Wang tiles.

Contents

Preface

In 1959 Feynman gave a visionary talk *There's plenty of room at the bottom* [7] where he anticipates that future computers will work in molecular scales. Many years later in 1994 Adleman conducted a real experiment [2] where he used DNA molecules to solve an NP-complete problem – Hamiltonian Path Problem. DNA computation became popular because it promised an advantage of extreme parallelism in solving hard computational problems.

Unlike initial excitement it showed to be very hard to use for practical problems, DNA computations still do too many errors. Many researchers are trying hard to decrease the amount of errors, many others are studying DNA computations from mathematical informatics point of view which is also the case of this thesis.

Work overview

Chapter 1. In this chapter we describe basic DNA principles and some fundamental concepts of mathematical informatics focusing on classes of formal languages from resource consumption point of view. Then we describe some theoretical models for specific DNA molecules together with their computer science properties. Similarly to the time and space resources of a Turing machine we present four kinds of resources of DNA computation and give a relation to resources of a Turing machine.

Chapter 2. This chapter presents a derived model suitable for solving NP-complete problems. The model is based on an existing formal model, on the other hand it can be implemented with real molecules. Two NP-complete problems and one NP problem are then solved using this model, one of them is also simulated using a DNA tile assembly simulator.

Notation and used symbols

$\mathbb{N}$	Positive integers, i.e. $\{1, 2, \ldots\}$.
$\mathbb{N}_0$	Non-negative integers, i.e. $\{0, 1, 2, \ldots\}$.
$\mathcal{P}(\mathcal{S})$	Power set of a set S.
$\mathbb{P}(E)$	Probability of an event E.

Chapter 1

Introduction to DNA computation

1.1 Basic DNA principles

DNA (deoxyribonucleic acid) is a large biomolecule carrying genetic information of living organisms. Its most common structure is the well-known double-helix which consists of two strands connected by hydrogen bonds. These strands are biopolymers built up by *polymerase chain reaction* (PCR) from small units – nucleotides. Each nucleotide consists of two parts: nitrogenous base and backbone molecules.

Nitrogenous bases. There are 4 nitrogenous bases in DNA: Adenine (**A**), Thymine (**T**), Cytosine (**C**) and Guanine (**G**). These molecules are responsible for making hydrogen bonds between strands in a manner following the Watson-Crick complementarity: only **A** – **T** and **C** – **G** pairs can be formed.

Backbone molecules. Backbone of a DNA strand is made of alternating deoxyriboses and phosphates. Phosphates only hold adjacent deoxyriboses, each deoxyribose moreover holds one nitrogenous base. Due to the deoxyribose carbon numbering, DNA backbone has so-called $5'$ and $3'$ ends, default reading order is $5' \to 3'$. DNA strands must be antiparallel so that nucleobases can connect.

1.2 Complexity and languages

1.2.1 Big O and other notations

Let us briefly remind O-, o-, Ω-, ω- and Θ-notations for functions on positive integers. We will denote $(\exists n_0 \in \mathbb{N})(\forall n > n_0)$ shortly by $(\forall^* n)$ which can be read "for almost all n". Also $(\forall n_0 \in \mathbb{N})(\exists n > n_0)$ will be denoted by $(\exists^\infty n)$ which can be read "there exist infinitely many n".

Definition 1.1. Let $f, g : \mathbb{N} \to \mathbb{N}$.

$$\begin{aligned}
O(f) &= \big\{\, g \;\big|\; (\exists C > 0)(\forall^* n)\big(g(n) < Cf(n)\big)\big\}\\
\omega^{(1)}(f) &= \big\{\, g \;\big|\; (\forall C > 0)(\exists^\infty n)\big(g(n) > Cf(n)\big)\big\}\\
\omega^{(2)}(f) &= \big\{\, g \;\big|\; (\forall C > 0)(\forall^* n)\big(g(n) > Cf(n)\big)\big\}\\
\Omega^{(1)}(f) &= \big\{\, g \;\big|\; (\exists C > 0)(\exists^\infty n)\big(g(n) > Cf(n)\big)\big\}\\
\Omega^{(2)}(f) &= \big\{\, g \;\big|\; (\exists C > 0)(\forall^* n)\big(g(n) > Cf(n)\big)\big\}\\
o(f) &= \big\{\, g \;\big|\; (\forall C > 0)(\forall^* n)\big(g(n) < Cf(n)\big)\big\}\\
\Theta(f) &= \big\{\, g \;\big|\; (\exists C_1, C_2 > 0)(\forall^* n)\big(C_1 f(n) \le g(n) \le C_2 f(n)\big)\big\}\\
g \sim f &\Leftrightarrow \lim_{n\to\infty} \frac{g(n)}{f(n)} = 1
\end{aligned}$$

The notation $g \sim f$ is an abbreviation for f *is asymptotically equivalent to* g.

Remark 1.1. Note that there are two distinct definitions for omegas. The class $\Omega^{(1)}$ is equivalent to the original definition introduced by Hardy [9] which states

$$g \in \Omega(f) \Leftrightarrow \limsup_{n\to\infty} \frac{g(n)}{f(n)} > 0.$$

The second class $\Omega^{(2)}$ was introduced by Knuth for good reasons described in [11]. In similar manner there are two definitions for ω.

Remark 1.2. Note that there are also some relations: the condition for $\Omega^{(1)}$ is negation of the condition for o so the sets $\Omega^{(1)}(f)$ and $o(f)$ are complementary for a given function f, similarly for $\omega^{(1)}$ and O. For the second variant one can easily check that $f \in \Omega^{(2)}(g) \Leftrightarrow g \in O(f)$ and $f \in \omega^{(2)}(g) \Leftrightarrow g \in o(f)$. There is also an equivalent condition for Θ:

$$g \in \Theta(f) \Leftrightarrow \big(g \in O(f) \,\wedge\, f \in O(g)\big) \Leftrightarrow \big(g \in O(f) \,\wedge\, g \in \Omega^{(2)}(f)\big).$$

Remark 1.3. The condition for $g \sim f$ can be easily seen to be equivalent to $|f - g| \in o(g)$. In Chapter 2 we will be mostly interested in this relation because it specifies the function better than Θ. For example, $2n \in \Theta(n)$ but $2n \not\sim n$.

1.2.2 Languages

Definition 1.2. Let Σ be a nonempty and finite set of *characters* which will be referred to as *alphabet*. Let us define the set of all *words* over alphabet Σ as $\Sigma^* = \bigcup_{n\in\mathbb{N}_0} \Sigma^n$ and the set of all nonempty words as $\Sigma^+ = \bigcup_{n\in\mathbb{N}} \Sigma^n$. The empty word will be denoted by ε. A *language* $\mathcal{L}$ over alphabet Σ is an arbitrary set of words: $\mathcal{L} \subseteq \Sigma^*$. The *complement* of a language $\mathcal{L}$ will be denoted by $\overline{\mathcal{L}} = \Sigma^* \setminus \mathcal{L}$.

Let us briefly remind Chomsky hierarchy of languages generated by generative grammars. Generative grammar is a fourtuple $G = (N, \Sigma, R, I)$ where N stands for *alphabet of nonterminals*, Σ stands for *alphabet of terminals* while $N \cap \Sigma = \emptyset$, $R \subseteq \big((N \cup \Sigma)^* \setminus \Sigma^*\big) \times (N \cup \Sigma)^*$ stands for rewriting grammar rules and $I \in N$ stands for *initial symbol.*

A rewriting grammar rule $(\xi, \eta) \in R$ is often written as $\xi \to \eta$ and it means the following: if you have a word $\alpha \in (N \cup \Sigma)^+$, $\alpha = \kappa\xi\lambda$ and $\xi \to \eta \in R$, you can rewrite α to $\kappa\eta\lambda$, denoted by $\alpha \vdash \kappa\eta\lambda$. Then $\mathcal{L}(G) = \big\{\alpha \in \Sigma^* \bigm| (\exists n \in \mathbb{N})\big(\exists \xi_1, \ldots, \xi_n \in (N \cup \Sigma)^+\big)(I \vdash \xi_1 \vdash \ldots \vdash \xi_n = \alpha)\big\}$ stands for a language generated by the generative grammar G. If we restrict grammar rules in the following manner we get Chomsky hierarchy of languages.

Recursively enumerable (type 0). Generated by unrestricted grammar rules.

Context-sensitive (type 1). Grammar rules are either all of the form $\alpha A\beta \to \alpha\eta\beta$ where α, $\beta \in (N \cup \Sigma)^*$, $A \in N$ and $\eta \in (N \cup \Sigma)^+$. Note that η cannot be empty. Or, if we assume that these rules do not have I on the right side, then the rule $I \to \varepsilon$ is also allowed.

Context-free (type 2). All grammar rules are of the form $A \to \eta$ where $A \in N$ and $\eta \in (N \cup \Sigma)^*$. Note that η can be empty.

Regular (type 3). Grammar rules are either all of the forms $A \to bB$ or $A \to b$ where A, $B \in N$ and $b \in \Sigma$. Or, if we assume that these rules do not have I on the right side, then the rule $I \to \varepsilon$ is also allowed.

Theorem 1.1. Let us denote L_i the set of languages of type i. Then $\mathsf{L}_3 \subsetneq \mathsf{L}_2 \subsetneq \mathsf{L}_1 \subsetneq \mathsf{L}_0$.

Remark 1.4. Note that given an alphabet Σ the set of all words Σ^* is countable. Moreover, one can sort Σ^* first by word length, then lexicographically, thus it is easy to define desired bijection $f : \mathbb{N} \leftrightarrow \Sigma^*$.

Formal language $\mathcal{L}$ can then be viewed as a Boolean function $g : \mathbb{N} \to \{0, 1\}$ defined as $g(n) = 1 \Leftrightarrow f(n) \in \mathcal{L}$.

1.2.3 P, NP and other classes

This section describes a few classes of languages from the resource consumption point of view. There exist many equivalent definitions, the following are taken from [4]. See also [4, Chapter 1.5.1] for a discussion reasoning about the importance of the class P.

Definition 1.3. $\mathcal{L} \subseteq \Sigma^*$. $\mathcal{L} \in \mathsf{P} \Leftrightarrow \big(\exists \text{ polynomial } p\big)\big(\exists \text{ deterministic Turing machine } M\big)$ $\big(\forall x \in \Sigma^*\big)\big(x \in \mathcal{L} \Leftrightarrow M \text{ accepts } x \text{ at most in } p(|x|) \text{ steps}\big)$.

Definition 1.4. $\mathcal{L} \subseteq \Sigma^*$. $\mathcal{L} \in \mathsf{NP} \Leftrightarrow \big(\exists \text{ polynomials } p, q\big)\big(\exists \text{ deterministic Turing machine } M\big)$ $\big(\forall x \in \Sigma^*\big)\Big(x \in \mathcal{L} \Leftrightarrow \big(\exists y \in \Sigma^{q(|x|)}\big)\big(M \text{ accepts } (x, y) \text{ at most in } p(|x| + |y|) \text{ steps}\big)\Big)$.
Such y will be referred to as *certificate* for x (with respect to $\mathcal{L}$ and M).

Definition 1.5. $\mathcal{L} \subseteq \Sigma^*$. $\mathcal{L} \in \text{co-}\mathsf{NP} \Leftrightarrow \big(\exists \text{ polynomials } p, q\big)\big(\exists \text{ deterministic Turing machine } M\big)$ $\big(\forall x \in \Sigma^*\big)\Big(x \in \mathcal{L} \Leftrightarrow \big(\forall y \in \Sigma^{q(|x|)}\big)\big(M \text{ accepts } (x, y) \text{ at most in } p(|x| + |y|) \text{ steps}\big)\Big)$.

The second possibility is to first define general classes DTime, NTime, DSpace and NSpace and after that to use their special case. Remind how Non-deterministic Turing machine (NTM) differs from Deterministic Turing machine (DTM):

- it is allowed to have an ambiguous transition function,
- it has different terminal states: accepting q_{accept} and halting without accepting nor rejecting q_{halt},

- ○ we say that it accepts the input x if and only if there *exists* a sequence of decisions ending in q_{accept}, as a consequence it declines x if and only if *for every* sequence of decisions it reaches q_{halt}.

Definition 1.6. $\mathsf{DTime}\big(f(n)\big) = \big\{\mathcal{L} \text{ language} \;\big|\; (\exists \text{ deterministic Turing machine } M)$ $(\forall x \in \Sigma^*)\big(x \in \mathcal{L} \Leftrightarrow M \text{ accepts } x \text{ in time } \le f(|x|)\big)\big\}$.
NTime, DSpace and NSpace are defined in a very similar manner: time is replaced with space, deterministic is replaced with non-deterministic.

Theorem 1.2. $\mathsf{P} = \bigcup\limits_{k\in\mathbb{N}} \mathsf{DTime}(n^k)$ and $\mathsf{NP} = \bigcup\limits_{k\in\mathbb{N}} \mathsf{NTime}(n^k)$.

Theorem 1.3. $\mathcal{L} \in \mathsf{co\text{-}NP} \Leftrightarrow \overline{\mathcal{L}} \in \mathsf{NP}$.

Note 1.5. Theorems 1.2 and 1.3 can give equivalent definitions of P, NP and co-NP.

Note 1.6. $\mathsf{NP} \subseteq \mathcal{P}(\Sigma^*)$ where $\mathcal{P}(S)$ denotes power set of the set S, so the notation co-NP might be confusing. Note that co-NP is *not* a complement to NP.

Remark 1.7. In case of P there is no “co” version. It follows from $\mathcal{L} \in \mathsf{P} \Leftrightarrow \overline{\mathcal{L}} \in \mathsf{P}$. This can be easily seen because deterministic Turing machine is capable of negation with the same time resources greater than n, non-deterministic is not known to. Note that Immerman–Szelepcsényi theorem states possibility of non-deterministic Turing machine negation in a limited *space*: $\mathcal{L} \in \mathsf{NSpace}(s(n)) \Leftrightarrow \overline{\mathcal{L}} \in \mathsf{NSpace}(s(n))$ for $s(n) \ge \log(n)$.

Definition 1.7. We say that a language $\mathcal{L}_1$ is *polynomial-time Karp reducible* to $\mathcal{L}_2 \Leftrightarrow \exists$ polynomial-time computable function $f : \Sigma^* \to \Sigma^*$ such that $(\forall x \in \Sigma^*)(x \in \mathcal{L}_1 \Leftrightarrow f(x) \in \mathcal{L}_2)$. This relation is denoted by $\mathcal{L}_1 \propto \mathcal{L}_2$.

Definition 1.8. $\mathcal{L} \in \mathsf{NP}\text{-hard} \Leftrightarrow (\forall \mathcal{L}' \in \mathsf{NP})(\mathcal{L}' \propto \mathcal{L})$.

Definition 1.9. $\mathcal{L} \in \mathsf{NP}\text{-complete} \Leftrightarrow \mathcal{L} \in \mathsf{NP} \cap \mathsf{NP}\text{-hard}$.

Note 1.8. If we found a polynomial-time algorithm for *any* NP-complete language, it would hold $\mathsf{P} = \mathsf{NP}$ which is not believed to be true. Thus assuming that $\mathsf{P} \subsetneq \mathsf{NP}$, there does not exist any deterministic polynomial algorithm for any NP-complete language.

Example 1.1. Remind some popular NP-complete problems: Boolean Formula Satisfiability (SAT), Hamiltonian Path Problem (HPP), Graph 3-coloring, Graph k-independent Set, Graph k-clique, Graph k-vertex Cover, Subset Sum and many others.

On the other hand, there are a few interesting NP problems which are not known to belong either to P or to NP-complete, one of them is Graph Isomorphism Problem.

Before we define probabilistic classes of languages (BPP, RP, co-RP and ZPP) we will remind the concept of Probabilistic Turing machine (PTM). As in the case of NTM, it is allowed to have an ambiguous transition function, moreover, in every state a transition probability is defined and it can have another terminal state q_{reject}.

Let M be a PTM. We say that M

- ○ decides language $\mathcal{L}$ if and only if for every $x \in \Sigma^*$ the probability of halting in the correct state (i.e. q_{accept} for $x \in \mathcal{L}$ and q_{reject} for $x \notin \mathcal{L}$) is greater than $2/3$,
- ○ decides language $\mathcal{L}$ in time $t(n)$ if and only if M decides language $\mathcal{L}$ and for every $x \in \Sigma^*$ it halts in time $\le t(|x|)$ regardless its randomness.

Now we can define those probabilistic classes. Note that to define the classes, we can again choose from two equivalent characterizations.

Definition 1.10. $\mathsf{BPTime}\big(f(n)\big) = \big\{\mathcal{L} \text{ language} \bigm| (\exists \text{ probabilistic Turing machine } M)$ $\big(M \text{ decides } \mathcal{L} \text{ in time } \leq f(|x|)\big)\big\}$.

Definition 1.11. $\mathsf{BPP} = \bigcup\limits_{k\in\mathbb{N}} \mathsf{BPTime}(n^k)$.

Definition 1.12. $\mathsf{RTime}\big(f(n)\big) = \Big\{\mathcal{L} \text{ language} \Bigm| (\exists \text{ probabilistic Turing machine } M)$ $(\forall x \in \Sigma^*)\Big(M \text{ halts in time } \leq f(|x|) \wedge \big(x \in \mathcal{L} \Rightarrow \mathbb{P}(M \text{ accepts } x) \geq 2/3\big) \wedge$ $\big(x \notin \mathcal{L} \Rightarrow \mathbb{P}(M \text{ accepts } x) = 0\big)\Big)\Big\}$.

Definition 1.13. $\mathsf{RP} = \bigcup\limits_{k\in\mathbb{N}} \mathsf{RTime}(n^k)$.

Definition 1.14. $\mathcal{L} \in \mathsf{co\text{-}RP} \Leftrightarrow \overline{\mathcal{L}} \in \mathsf{RP}$.

Remark 1.9. BPP allows PTM to make both wrong decisions (for $x \in \mathcal{L}$ as well as for $x \notin \mathcal{L}$), on the other hand RP does not allow any error for $x \notin \mathcal{L}$ and co-RP does not allow any error for $x \in \mathcal{L}$. The following class ZPP (from *zero error*) does not allow any error, it only allows halting neither with accepting nor with rejecting.

Definition 1.15. $\mathsf{ZPTime}\big(f(n)\big) = \Bigg\{\mathcal{L} \text{ language} \Bigg| (\exists \text{ probabilistic Turing machine } M\Big)$ $\Big(\forall x \in \Sigma^*\Big)\Bigg(M \text{ halts in time } \leq f(|x|) \wedge$ $\Big(x \in \mathcal{L} \Rightarrow \big(\mathbb{P}(M \text{ accepts } x) \geq 2/3 \wedge \mathbb{P}(M \text{ rejects } x) = 0\big)\Big) \wedge$ $\Big(x \notin \mathcal{L} \Rightarrow \big(\mathbb{P}(M \text{ rejects } x) \geq 2/3 \wedge \mathbb{P}(M \text{ accepts } x) = 0\big)\Big)\Bigg)\Bigg\}$.

Definition 1.16. $\mathsf{ZPP} = \bigcup\limits_{k\in\mathbb{N}} \mathsf{ZPTime}(n^k)$.

The following theorems state some properties and relations between the classes.

Theorem 1.4.

$$\mathcal{L}_1, \mathcal{L}_2 \in \mathsf{ZPP} \Rightarrow \overline{\mathcal{L}}_1,\ \mathcal{L}_1 \cap \mathcal{L}_2,\ \mathcal{L}_1 \cup \mathcal{L}_2 \in \mathsf{ZPP}.$$

Theorem 1.5.

$$\begin{aligned} \mathsf{P} \subseteq \mathsf{ZPP} &= \mathsf{RP} \cap \mathsf{co\text{-}RP}, \\ \mathsf{RP} &\subseteq \mathsf{NP} \cap \mathsf{BPP}, \\ \mathsf{co\text{-}RP} &\subseteq \mathsf{co\text{-}NP} \cap \mathsf{BPP}. \end{aligned}$$

Conjecture 1.6.

$$\begin{aligned} \mathsf{P} &\subsetneq \mathsf{NP}, \\ \mathsf{P} &= \mathsf{BPP}. \end{aligned}$$

1.3 Strand models

There exist or can be synthesized many types of molecules, well described by Winfree [14] even with their inception reaction. The most important structures are linear strands (Sections 1.3.1, 1.3.2, Figure 1.1), dendrimer structures (Section 1.3.3, Figure 1.4) and double crossover molecules (Section 1.3.4, Figure 1.5).

Note 1.10. DNA naturally forms double-helices which would be confusing in figures so there will mostly appear schemes of "untwisted" molecules, only double crossover molecules will be described more precisely.

1.3.1 Linear strands

Linear strands are just a piece of a double-helix with two types of ends: either with one strand longer than the other (sticky end) or with both strands connected to each other. The top molecule in Figure 1.1 is called *hairpin*. Winfree [15] proved that restricted class of linear strands is equivalent to regular languages, this means

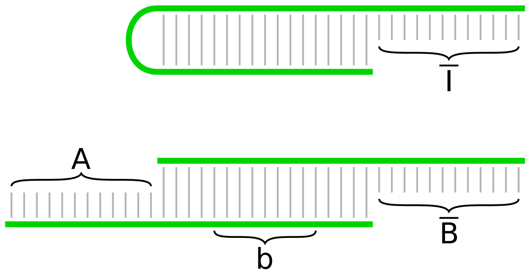


Figure 1.1: Linear strands: a strand for initial symbol I and a strand for rule $A \to bB$.

- given a set of linear strands, the generated language is regular,
- given a regular language there exists a set of linear strands which generate it.

Figure 1.1 shows how to assign grammar rules to linear strands.

1.3.2 Adleman's experiment

Linear strands were used in Adleman's ground-breaking work [2]. He showed that DNA molecules are really capable of computation. He exploited the huge parallelism in DNA computation for one of the most fundamental NP-complete problems – the Hamiltonian Path Problem (HPP) in a directed graph with designated vertices v_{begin} and v_{end}.

Let us remind this type of HPP. Given a directed graph G_n with n vertices and two designated vertices v_{begin} and v_{end}, the problem is to answer whether there exists an oriented path from v_{begin} to v_{end} through the graph such that the path visits every vertex. Note that *path* cannot visit any vertex more than once from definition.

Adleman originally used a graph with seven vertices shown in Figure 1.2. It can be seen that the path $0 \to 1 \to 2 \to 3 \to 4 \to 5 \to 6$ is Hamiltonian[1].

[1] Note that it can be relabelled in such a nice way without loss of generality.

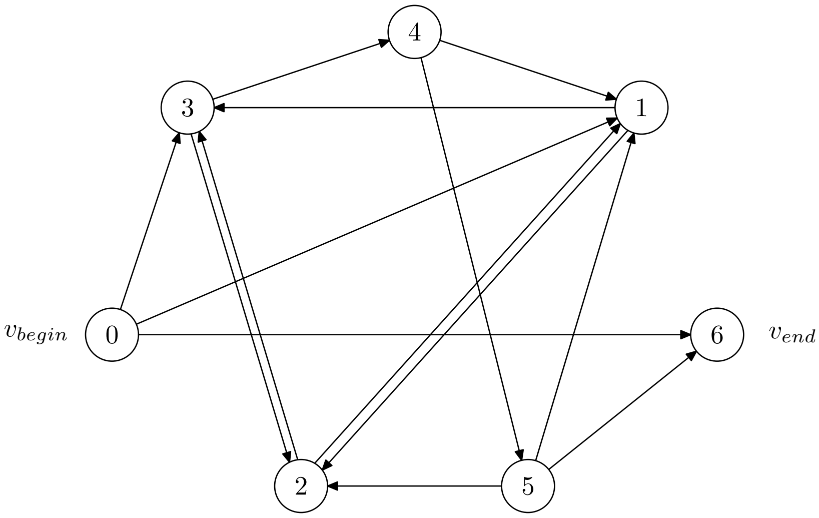


Figure 1.2: Adleman's original graph.

Adleman first presented the following non-deterministic five-step algorithm, whose steps were described in his work in terms of DNA manipulations:

Step 1. Generate random paths through the graph.

Step 2. Keep only those paths that begin with v_{begin} and end with v_{end}.

Step 3. If the graph has n vertices, then keep only those paths that enter exactly n vertices.

Step 4. Keep only those paths that enter all of the vertices of the graph at least once.

Step 5. If any path remains, say "Yes"; otherwise, say "No."[2]

To see how DNA can compute, let us describe this example more precisely. The computation itself (meaning the inception of the final solution) is hidden in Step 1. Each vertex i is associated with a random[3] 20-mer sequence of DNA, let us denote its $5' \to 3'$ orientation by O_i, its 10-mer prefix by p_i and its 10-mer suffix by q_i. Each edge $i \to j$ is then associated with $O_{i\to j} = \overline{q_i p_j}$ sequence with reverse backbone orientation where $\overline{a}$ stands for Watson-Crick complementary word to the word a. There is an exception for $i = v_{begin}$ and every k: instead of $\overline{q_i p_k}$ there is $\overline{O_i p_k}$ and in a similar way for $j = v_{end}$ and $\overline{q_k p_j}$.

$$\begin{array}{lccccr}
 & & \multicolumn{2}{c}{O_{i\to j}} & & \\
3' \leftarrow & & \ldots|\texttt{CTTTCGAATA} & \texttt{AAGATCAGTC}|\ldots & & \leftarrow 5' \\
5' \to & \ldots|\texttt{CCTCTCGCGA} & \texttt{GAAAGCTTAT}| & \texttt{TTCTAGTCAG} & \texttt{CACTCTTTGT}|\ldots & \to 3' \\
 & \multicolumn{2}{c}{O_i} & \multicolumn{2}{c}{O_j} &
\end{array}$$

Figure 1.3: Example of assigned sequences.

[2] This is the original version, I would rectify the fifth step: If any paths remain, say "Yes"; otherwise say "*I do not know*". That is because it may happen that there exists a valid path but unfortunately it did not assemble or got lost. Note the similarity to NP versus co-NP, see Section 1.2.3.

[3] We will expect those sequences to be different enough.

It can be easily seen that all correctly bonded double-strands correspond with a valid walk through G_n. Moreover, all double-strands without sticky ends represent a valid walk from v_{begin} to v_{end} through G_n.

All the other steps are fully described in [2]. The most important thing is that the most time-demanding step is Step 4. In this step one has to purify[4] the product of Step 3. This process consequently extracts for every vertex i only those DNA strands which contain a substring representing vertex i. Thus this algorithm requires $O(n)$ laboratory steps which we will later consider unfeasible. Better solution with $O(1)$ laboratory steps was introduced by Winfree [15].

1.3.3 Dendrimer structures

Dendrimer structures are multi-strand molecules which form trees, see Figure 1.4. There can be arbitrary[5] number of ends of both types, see Figure 1.1. Winfree [15] proved similar property as for linear strands: dendrimer structures are equivalent to context-free languages. Figure 1.4 shows an example of a context-free rule.

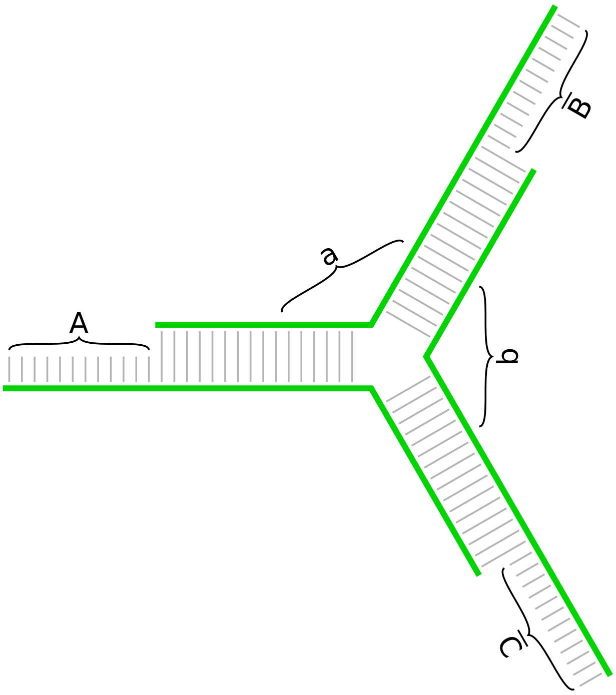


Figure 1.4: Dendrimer structure for rule $A \to aBbC$.

[4] The purifying procedure is so called *biotin-avidin magnetic beads system.*

[5] At this moment the model is theoretical though practically it has limitations. Note that even with limited number of branches the size of the whole system must be bounded: having $O(2^n)$ molecules, they must fit in space $O(n^3)$ thus n must be bounded.

1.3.4 Double crossover molecules

Double crossover molecules are the most important ones. Though they are more complicated they are still very rigid, see [8]. Moreover they are theoretically capable of universal computation, see [15]. As we mentioned in Note 1.10 we will also describe their inner structure.

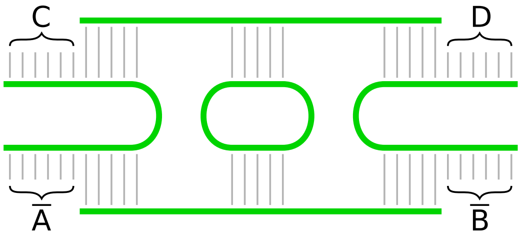
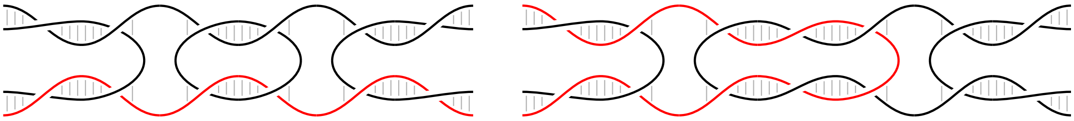


Figure 1.5: Double crossover molecule scheme.

There are many possibilities how those strands can be twisted and connected. The most common ones are DAE and DAO molecules (Double-crossover, Antiparallel helical strands, Even or Odd, respectively, number of half-turns between crossovers), see Figure 1.6. It can be seen that DAO molecules form tilings with strands jumping from one stage to another, on the other hand DAE molecules form tilings with a strand leading through entire stage in a row.



(a) DAE.

(b) DAO.

Figure 1.6: DAO vs. DAE.

In a real computation we will need to read the bottom line's sequence in a row thus it is practically reasonable to use DAE molecules. Moreover we will require even number of half-turns between crossovers in adjacent molecules. This is well explained in [15, p.37].

DAE molecules thus have many reasons to be the most promising thus Chapter 2 will be dedicated especially to models with these molecules.

1.4 Wang-tile models

1.4.1 Definition

Wang tiles are named after Hao Wang who introduced this concept in his work *Proving Theorems by Pattern Recognition*, [13]. His original definition was rather informal but there appeared also some very formal definitions, e.g. Rothemund, Winfree, [12]. We will rather define Wang tile models less formally because it will be clear what they mean and excessive formality could cause confusions.

Wang tile is a square tile with one color (also index, glue) from a finite nonempty set I on each its edge[6]. Let us denote (nonempty) set of all available tiles T. Wang tiling is a mapping $\mathcal{T} : (\mathbb{Z}^2) \to T$ with restrictions: neighboring tiles must have the same color on the adjacent edge and its domain is required to be (topologically) connected. Wang tiling $\mathcal{T}_2$ is called *reachable* from Wang tiling $\mathcal{T}_1$ if and only if $\mathcal{T}_1 \subseteq \mathcal{T}_2$. A tiling $\mathcal{T}$ is called *terminal* if and only if there does not exist any strictly larger tiling reachable from $\mathcal{T}$.

Rothemund and Winfree [12] extended this definition for DNA tile assembly purposes and introduced Abstract Tile Assembly Model[7] (aTAM). Additionally

- every color c is associated with a nonnegative integer $g(c)$ which will also be referred to as *glue strength*,
- there exists an empty color denoted by ε which can be neighboring with arbitrary color,
- tiling is only allowed to grow
  - from a special initial tile denoted by t_0 from $(0, 0)$ (*initial configuration*),
  - one tile per step,
  - the sum of glues connected in one step must be greater than or equal to given integer treshold[8], so called *temperature*, denoted by τ.

Reachability relation will be restricted accordingly to these rules, *reachable in (tilesystem) T* will mean reachable from initial configuration, same for *terminal tiling in (tilesystem) T*.

Similarly to Turing machines, tilesystem T will be called:

Deterministic if and only if there exists unique terminal tiling in T;

Non-deterministic if there are no other restrictions;

Probabilistic if and only if every possible step in every stage has a defined probability. See the following note for our proposal how the probability could be defined.

Note 1.11. If there is a unique place and a unique tile the probability is 1. If there is a unique place but there are more candidate tiles, the probability $\mathbb{P}_C(t)$ of connecting tile t should be distributed to these tiles somehow according to their ratio of concentration (denoted by $\mathbb{P}_0(t)$) and their total connectible glue strength (denoted by $G(t)$). Most straightforward is weighted average:

$$\mathbb{P}_C(t) = \frac{G(t) \cdot \mathbb{P}_0(t)}{\sum\limits_{u \text{ possible tile}} G(u) \cdot \mathbb{P}_0(u)}.$$

In the most general case there are more places, each having several candidates. The probability $\mathbb{P}_C(t, i)$ of connecting tile t on place i can be defined as weighted average again, now with one more sum over all possible places:

$$\mathbb{P}_C(t, i) = \frac{G(t, i) \cdot \mathbb{P}_0(t)}{\sum\limits_{j \text{ possible place}} \sum\limits_{\substack{u \text{ possible tile} \\ \text{on place } j}} G(u, j) \cdot \mathbb{P}_0(u)}.$$

[6] Tiles keep orientation, they are *not* allowed to rotate nor reflect.

[7] A modification to this model introduced by Winfree is Kinetic Tile Assembly Model (kTAM) which describes system dynamics.

[8] Only small numbers (1 and 2) are interesting to study; there are different results in Section 1.4.3

1.4.2 Studied complexities

Definition 1.17. All the following complexities are considered as functions of size of the problem, which will be denoted by n:

Biostep complexity. Refers to the number of laboratory steps required to handle the computation, denoted by $Bs(n)$. Adleman [3] describes formally in his *unrestricted model* few types of such lab procedures – *Separate, Merge, Detect* and *Amplify*, Winfree [15] adds another – *Append*. Both of them remind that one biostep takes tens of minutes. Thus the only practically feasible DNA algorithms are those with $O(1)$ biostep complexity.

Binding complexity. Refers to the number of bindings in terminal tiling, denoted by $Bnd(n)$. In deterministic computation the value is unique, in non-deterministic computation we consider the value for the smallest accepting terminal tile, in probabilistic computation we consider the mean value. Too high binding complexity leads to lower probability of correct computation P_c because it holds $P_c(n) = (1-p_e)^{Bnd(n)} \approx 1 - p_e \cdot Bnd(n)$ where p_e denotes probability of erroneous binding.

Tile complexity. Refers to the number of different DNA tiles, denoted by $Ti(n)$. The higher tile complexity the more demanding it is to prepare required tiles.

Glue complexity. Refers to the number of different sticky-end sequences (commonly referred to as *glues*), denoted by $Gl(n)$. Each sequence with its Watson-Crick complement is considered as one glue. Higher glue complexity will require longer DNA sequences in the sticky ends.

Lemma 1.7.

1. $Ti(n) \leq Gl^4(n)$,
2. $Gl(n) \leq 4\,Ti(n)$.

Proof. For combinatoric reasons,

1. having $Gl(n)$ glues, we cannot make more than $Gl^4(n)$ different tiles,
2. having $Ti(n)$ tiles, there can appear at most $4\,Ti(n)$ glues.

□

Thesis 1.8. Feasible DNA algorithms comply $Bs(n) \in O(1)$; $Bnd(n)$, $Ti(n)$, $Gl(n) \in \bigcup_{k=0}^{\infty} n^k$.

1.4.3 Computational power

The most exciting thing about aTAM is that

- it is known to be capable of universal computation at temperature 2 in 2D,
- also at temperature 1 in 3D,
- but it is not known to be universal or not at temperature 1 in 2D[9].

[9] Universality has been reached only with modifications to the original model, see [5], [10].

Another interesting problem is “How many tile types are required to self-assemble an $n \times n$ square?” Let us show known results in the following table.

	$n \times n$ **squares**		**Computational**
	Lower bound	Upper bound	**Power**
$\tau = 2$, 2D	See [12], $\Theta(\frac{\log n}{\log \log n})$, see [1]		Universal, see [15]
$\tau = 1$, 3D	$\Omega(\frac{\log n}{\log \log n})$, see [12]	$O(\log n)$, see [6]	Universal, see [6]
$\tau = 1$, 2D	$\Omega(\frac{\log n}{\log \log n})$, see [12]	$2n - 1$, see [12]	Unknown

1.4.4 Turing universality of 2D tiles at $\tau = 2$

Here we propose an alternative and more straightforward 2D tilesystem denoted by $\mathcal{T}_{TM}$ which directly simulates Turing machine at $\tau = 2$ proving Turing universality of this model, see Figures 1.7 and 1.8. All tiles are described within figures.

Remark 1.12. This tilesystem can simulate deterministic Turing machine as well as non-deterministic or probabilistic if we consider corresponding tilesystem, all in $O(1)$ biosteps.

Note that the worst case occurs when the head is changing its step direction because all the rest of the tape must be copied. Following lemma gives upper bound for binding complexity as well as for the other studied complexities.

Lemma 1.9. Studied complexities in tilesystem $\mathcal{T}_{TM}$ are bounded as follows:

Biostep. $Bs(n) \in O(1)$.

Binding. $Bnd(n) \in O\big(s(n) \cdot t(n)\big)$ where $t(n)$ stands for time and $s(n)$ for space required by the simulated Turing machine.

Tile. $Ti(n) \in O(n)$.

Glue. $Gl(n) \in O(n)$.

Proof.

Biostep. All tiles are designed to operate altogether thus only constant number of biosteps is needed.

Binding. All Turing machine steps are simulated one-to-one or one-to-constant number of bindings with only one exception which is tape-copying as soon as head changes its direction. The length of used tape is less than or equal to $s(n)$. Every copied length is thus bounded by $s(n) + C$ where C is a constant, copying process is thus bounded by $2(s(n) + C)$ bindings. Copying occurs maximally once per simulated step thus number of copying is bounded by $t(n)$. Altogether number of bindings is bounded by $2(s(n) + C) \cdot t(n) \in O\big(s(n) \cdot t(n)\big)$.

Tile. Number of non-input tiles is constant, it is proportional to Turing machine size which is constant. There must only be prepared n special input tiles thus $Ti(n) \in O(n)$.

Glue. Follows from Lemma 1.7 which states that $Gl(n) \leq 4Ti(n)$.

□

Corollary 1.10. In 2D at temperature $\tau = 2$ using corresponding type of aTAM (i.e. deterministic, non-deterministic or probabilistic), all classes resistant to polynomial slowdown (P, ZPP, RP, BPP, NP, ...) remain preserved with respect to all studied complexities. Moreover, biostep complexity remains in $O(1)$.

1.4.5 Feasibility of the class BPP

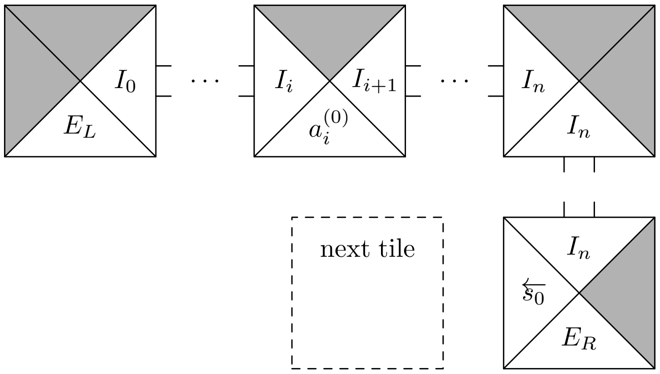
Remind that every language from BPP is decidable on a PTM in polynomial time which means that probability of correct result is greater than $^2/_3$. Note that one can reach probability of correct result higher than arbitrary constant smaller than 1 by constant number of iterations. Thus it is reasonable to consider BPP as practically feasible.

Here we introduce a similar proposal for feasibility of DNA algorithms. In Definition 1.17 we proposed that biostep complexity of feasible algorithms must comply $Bs(n) \in O(1)$ which is proved in Lemma 1.9 for Turing universal tilesystem $\mathcal{T}_{TM}$. In Thesis 1.8 we proposed that other studied complexities must be polynomial and Corollary 1.10 states preserving of classes P, ZPP, RP, BPP, NP, ... in tilesystem $\mathcal{T}_{TM}$ at temperature $\tau = 2$. Note 1.11 proposes how a probabilistic aTAM could be practically simulated. These arguments altogether justify Corollary 1.11.

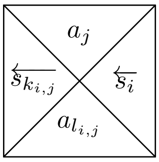
Corollary 1.11. BPP is feasible in 2D at temperature $\tau = 2$.

Note 1.13. Remind that P, ZPP, RP and co-RP $\subseteq$ BPP.

Assembly of input tape. E_L is left stop, $a_i^{(0)}$ is i-th symbol of input word $a_0^{(0)} \dots a_{n-1}^{(0)}$.
Following assembly starts at a place denoted "next tile" while simulating a Turing machine reading $a_{n-1}^{(0)}$ and being in state s_0. The arrow over s_0 means "comes from right".



Coming from right, being in state s_i, reading tape symbol a_j.
Transition function says: write $a_{l_{i,j}}$, switch to state $s_{k_{i,j}}$ and go left.



Situation is like before with only difference: go right.
Now the rest of the tape must be copied by special tiles which thus exist for all pairs a_m, $s_{k_{i,j}}$ of tape symbol and state, respectively.

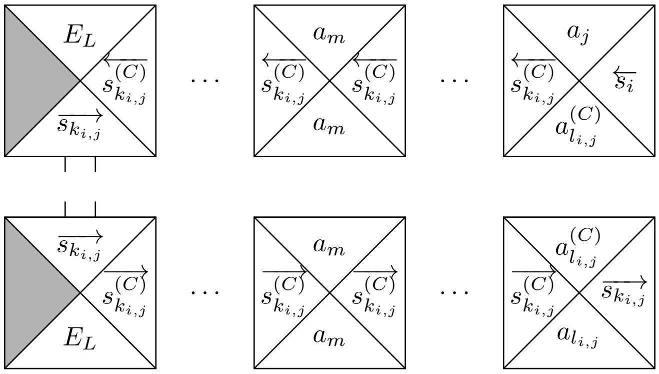
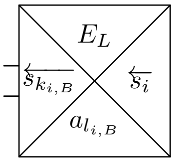
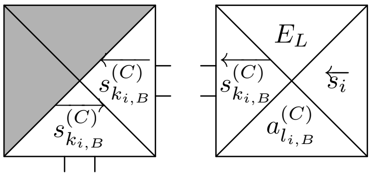

E_L
$\overleftarrow{s_{k_{i,j}}^{(C)}}$
$\overrightarrow{s_{k_{i,j}}}$
$\dots$
a_m
$\overleftarrow{s_{k_{i,j}}^{(C)}}$
$\overleftarrow{s_{k_{i,j}}^{(C)}}$
a_m
$\dots$
a_j
$\overleftarrow{s_{k_{i,j}}^{(C)}}$
$\overleftarrow{s_i}$
$a_{l_{i,j}}^{(C)}$
$\overrightarrow{s_{k_{i,j}}}$
$\overrightarrow{s_{k_{i,j}}^{(C)}}$
E_L
$\dots$
a_m
$\overrightarrow{s_{k_{i,j}}^{(C)}}$
$\overrightarrow{s_{k_{i,j}}^{(C)}}$
a_m
$\dots$
$a_{l_{i,j}}^{(C)}$
$\overrightarrow{s_{k_{i,j}}^{(C)}}$
$\overrightarrow{s_{k_{i,j}}}$
$a_{l_{i,j}}$

Figure 1.7: Tileset $\mathcal{T}_{TM}$, 1 of 2.

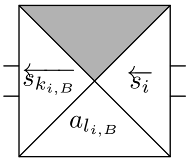
Coming from right, being in state s_i, reading left stop E_L which stands for blank symbol in terms of tape alphabet.
Transition function says: write $a_{l_{i,B}}$, switch to state $s_{k_{i,B}}$ and go left. Note that a glue of strength 2 must have been used.



Situation is like before with only difference: go right.
Note that tiles from previous figure are utilized to finish copying.


$\overleftarrow{s^{(C)}_{k_{i,B}}}$
$\overrightarrow{s^{(C)}_{k_{i,B}}}$
E_L
$\overleftarrow{s^{(C)}_{k_{i,B}}}$
$\overleftarrow{s_i}$
$a^{(C)}_{l_{i,B}}$

Coming from right, being in state s_i, reading blank symbol.
Transition function says: write $a_{l_{i,B}}$, switch to state $s_{k_{i,B}}$ and go left.



Situation is like before with only difference: go right.
Note that there already exist tiles which finish the turnaround.

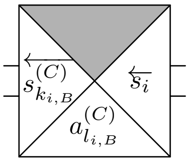


Figure 1.8: Tileset $\mathcal{T}_{TM}$, 2 of 2.

Chapter 2

Solutions to NP problems

First we define a simplifying model based on aTAM which was in fact originally used by Winfree [15] as an example of a tilesystem solving Hamiltonian Path Problem (HPP) with DAE molecules. Then we use this model in few examples of NP and NP-complete problems.

2.1 Abstract model for DAE units

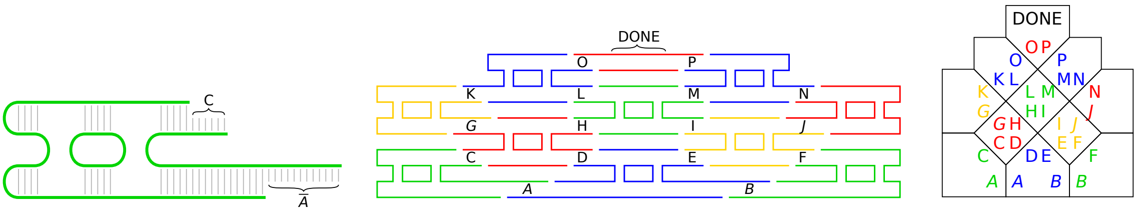
Definition 2.1. Let us define DaTAM (DAE aTAM) as aTAM where

- temperature is set to $\tau = 2$,
- there are another 5 tile shapes (even with their vertical reflections) denoted Bottom tile, Bottom corner tile, Border tile, Verification tile and DONE tile, see Figure 2.1c – tiles ADEB, CA, KG, KLO and OP DONE, respectively,
- initial tile t_0 is set to the left bottom tile (tile CA in Figure 2.1c),
- glue strengths are restricted to corresponding tile type, all of which appear and are described within Figure 2.1.

Note 2.1. DaTAM can be easily simulated by classical aTAM at temperature 2.

The bottom tiles have side glue strengths 2, thus they can form a linear sequence. This sequence plays the role of the seeked certificate from Definition 1.4. Then the other tiles "verify" correctness of this certificate with polynomial complexity. Note that binding complexity is related to probability of error thus we will be interested even in leading coefficients, see example in Remark 1.3.

For practical decoding reasons (see Winfree [15]) the sequences on the top of the bottom tiles must be physically present also wherever on the bottom DNA strand, see Figure 2.2.



(a) Corner DAE unit (b) Scheme of self-assembly (c) Abstract model

Figure 2.1: Evolution of DaTAM model from DAE units to tiles. Here glues A, B, G and J are of strength 2, all the other are of strength 1.

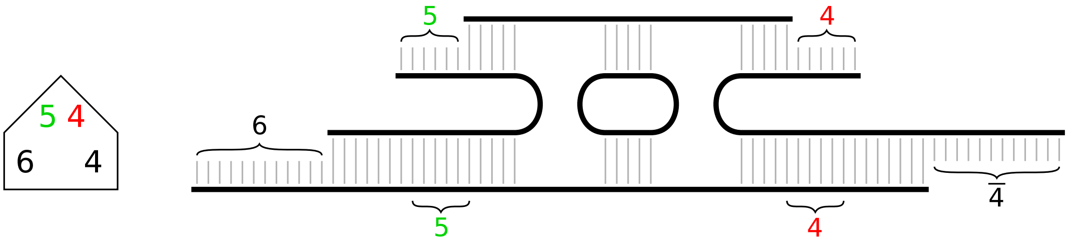


Figure 2.2: Bottom tile with desired sequences in the bottom strand.

2.2 k-clique Problem

k-clique Problem belongs to NP-complete problems, its task is to decide whether there exists a clique with k vertices in a given graph G with n vertices. Note that there exists a k-clique in G if and only if there exists a k-independent set in $\overline{G}$ if and only if there exists an $(n-k)$-vertex cover of $\overline{G}$ where $\overline{G}$ stands for the complement to G.

For technical reasons we need k to be even in all following problems; a reduction from odd numbers will be solved in every single problem.

Here if k is odd, we add another vertex connected to every original vertex resulting in G'. Then existence of a $(k+1)$-clique in G' is equivalent to existence of a k-clique in G. In the following example in Figure 2.3 we consider k even.

Set of tiles

Here we describe the tiles used by the proposed DNA algorithm, see Figure 2.3. The amount of required tiles of a given type will be given in terms of n – number of vertices, e – number of edges and k – size of seeked clique.

Bottom tiles. These tiles have colorless labels $2l$ and $2l-2$ $(0 < l \leq \frac{k}{2})$ on the bottom left and right sides, respectively. On the top sides there are numerically ordered[1] numbers

[1] Later we will order by color. Note that this restriction does not reduce the set of candidate k-cliques.

of vertices of all edges having $(k-2l+2)$-th and $(k-2l+1)$-th color, respectively. $e \cdot \frac{k}{2}$ tile types were required.

Bottom corner tiles. These two tiles have the largest and the least colorless number, respectively, on the bottom. On the top they have a special glue: # which can be viewed as $-\infty$ with respect to used ordering and * as $+\infty$, respectively. 2 tile types were required.

Inner tiles. These tiles are responsible for ordering[2] by color during which they verify existence of every edge. There exist all 2-color combinations of numbers of all connected vertices in both number-orders with both correct and reverse color-order on the bottom. On the top there are the same glues with correct color order. Moreover there exist similar tiles with sharp and asterisk instead of left or right number, respectively, with an exception: sharp and least color, and asterisk and largest color do not exist – these are reserved for verification tiles. Remind that the top sharp or asterisk must have glue strength equal to 2.

As soon as there appear numbers of unconnected vertices next to each other (in the following text also "meet"), the self assembly cannot continue and reach "DONE" because no tile can connect to this place. Note that colors were generated in reverse color order so they must meet each other during color ordering. Every missing edge would be revealed thus "DONE" tile connects if and only if the generated subset is a clique. $2 \cdot \binom{k}{2} \cdot 2e + 2(k-1)n \sim 2k^2e + 2kn$ tile types were required.

Border tiles. There are two tile types on the borders, one with sharps, one with asterisks. They just keep the structure growing up and signalize the border. Remind that the bottom sharp or asterisk must have glue strength 2. 2 tile types were required.

Verification tiles. As soon as the least color meets sharp and the largest color meets asterisk there connect two special tiles which start verification whether nothing is missing. There exist two types of verification sequences "C" and "D" with all color–number combinations: "D" with the smaller half (by color), "C" with the higher half. kn tile types were required.

DONE tile. If everything is verified and left and right verification sequences meet each other, "DONE" tile will be connected to signalize a correct solution. 1 tile type was required.

Summed up, the tile complexity of this DNA algorithm is asymptotically equivalent to $2k^2e + 3kn$. To compute the binding complexity easily we use the following lemma.

Lemma 2.1. In a square DaTAM tiling with n bottom tiles, the number of bonds is asymptotically equivalent to $4n^2$.

Proof. Note that the square can be divided by diagonals into four triangles consisting of asymptotically $\frac{n^2}{2}$ tiles thus there are asymptotically $2n^2$ tiles. Every inner tile has four bonds, every inner bond belongs to two tiles thus there are asymptotically $4n^2$ bonds. □

The binding complexity is thus asymptotically equivalent to $5/4 \cdot 4\left(\frac{k}{2}\right)^2 = 5/4\, k^2$, the glue complexity is asymptotically equivalent to kn.

[2] Principially they are the same as in Winfree [15].

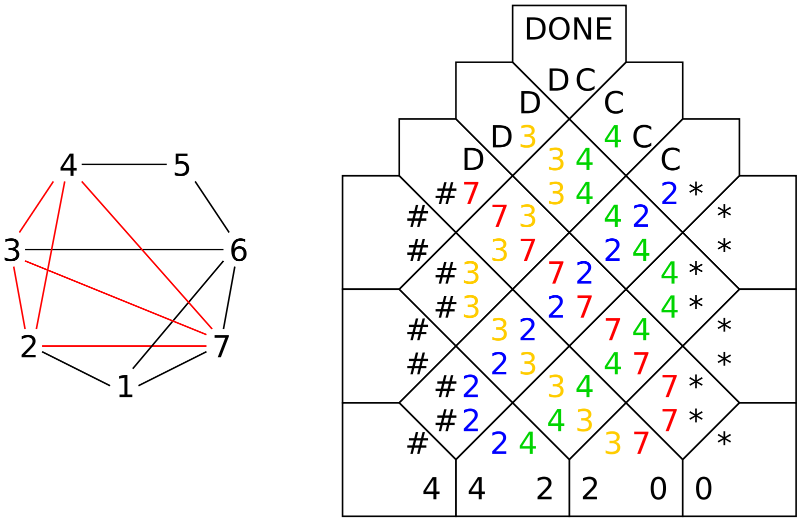


Figure 2.3: k-clique computation. Color order is defined by the wavelength of the colors.

2.2.1 Simulation in `xgrow`

`xgrow` is an open source DNA tile assembly simulator written by Erik Winfree, Rebecca Schulman and Constantine Evans and it is freely available on the internet[3]. It is capable of both aTAM and kTAM simulations.

`xgrow` is a program executable from command line. It takes settings and a file with tiles in input parameters, then it conducts a simulation. We used `xgrow` to verify our algorithm for k-clique Problem so we wrote a script which generates input file with tiles for `xgrow`. The script is called `k-clique.rb`, it is written in Ruby language and it is provided on the attached CD.

Running a simulation

Though everything is cross-platform, this paragraph describes how to run a simulation under Linux systems. You need to download and compile `xgrow` and install Ruby interpreter. Then you need to generate an input graph representation. Open Terminal in your directory with `k-clique.rb` and run `irb` which is an interactive Ruby shell. Then run following commands:

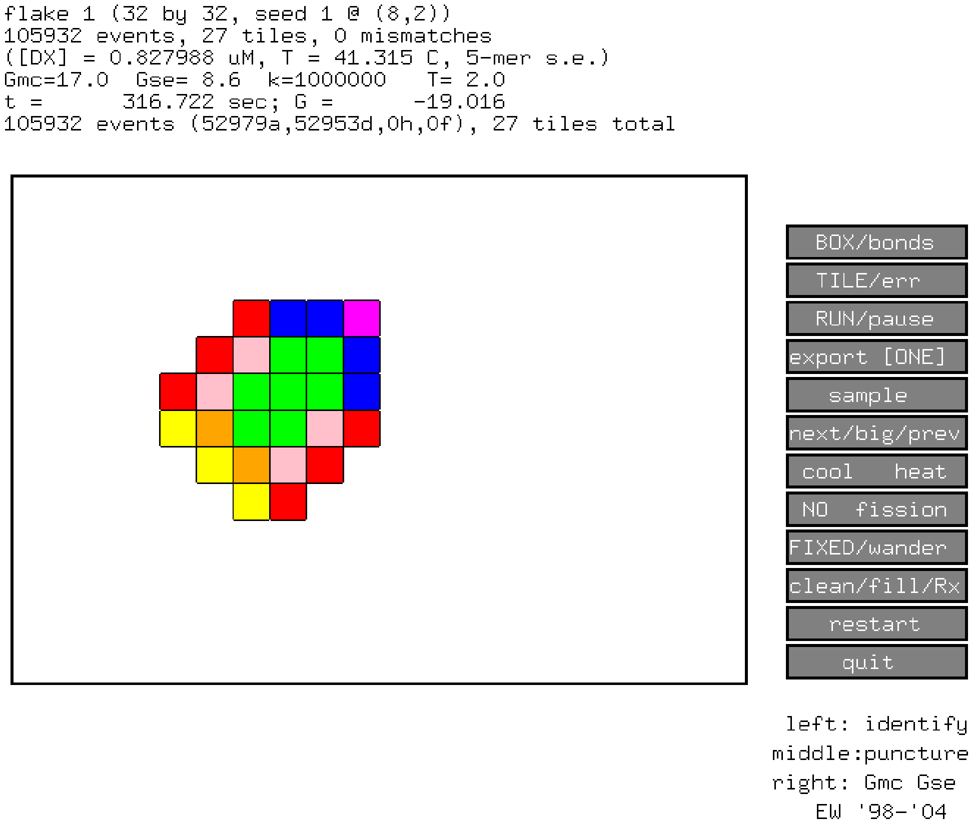
```
require "yaml"
# replace with adjacency matrix of your graph written
# as an array of lines (upper triangle suffices):
graph = [[0, 1, 0], [0, 0, 1], [0, 0, 0]]
# define output file
f = File.open("graph.yaml", "w")
# dump graph into yaml, write to file
f.write(YAML.dump(graph))
f.close
```

[3] `www.dna.caltech.edu/Xgrow` as of June 2014.

```
exit
```

Your graph is now stored in `graph.yaml`. Now run `k-clique.rb` in your directory with two parameters: first parameter is k – size of clique, second parameter is filename with your graph, and redirect output to a new `tiles` file, e.g. `./k-clique.rb 4 graph.yaml > 4-clique.tiles`. Then move your `tiles` file to `xgrow-install-dir/tilesets/` directory and run an aTAM simulation by e.g. `./xgrow 4-clique T=2` or a kTAM simulation by e.g. `./xgrow 4-clique`.

Note that `xgrow` performs a probabilistic simulation thus DONE tile[4] can appear very unlikely (depends on probability of random choosing a k-clique in your graph). So if you want to see an accepting computation you need to set probabilities of tiles which generate our clique to a reasonably high value. These probabilities can be set in your `tiles` file in the block `tile edges`, on every line there is a number in square brackets which represents the relative concentration which is proportional to probability. A sample output for aTAM simulation can be seen in Figure 2.4.



```
flake 1 (32 by 32, seed 1 @ (8,2))
105932 events, 27 tiles, 0 mismatches
([DX] = 0.827988 uM, T = 41.315 C, 5-mer s.e.)
Gmc=17.0  Gse= 8.6  k=1000000   T= 2.0
t =      316.722 sec; G =      -19.016
105932 events (52979a,52953d,0h,0f), 27 tiles total
```

Figure 2.4: k-clique accepting computation simulated by `xgrow` with aTAM. Magenta tile represents DONE tile.

Running a kTAM simulation in `xgrow` leads to a very unstable evolution process with many errors, it is very hard (but not impossible) to reach DONE tile. The only possibility to get a flake without errors is slow annealing which can be regulated by `cool heat` button.

[4] In our script the DONE tile has magenta color.

2.3 Graph 3-coloring Problem

Another NP-complete problem is Graph 3-coloring Problem, its task is to decide whether there exists an assignment of three colors to vertices of the graph so that no connected vertices have the same color.

If k is odd, we add a separated vertex resulting in graph G'. Then G' is 3-colorable if and only if the original graph G is 3-colorable. See example in Figure 2.5.

Set of tiles

Bottom tiles. These tiles have colorless labels $2l$ and $2l-2$ $(0 < l \leq \frac{k}{2})$ on the bottom left and right sides, respectively. On the top sides there are all color combinations of $2l-1$ and $2l-2$; if corresponding vertices are connected by edge, single-color combinations can be omitted. $\frac{9n}{2}$ tile types were required.

Bottom corner tiles. These tiles are exactly the same like for k-clique. 2 tile types were required.

Inner tiles. These tiles are responsible for numerical ordering, they are very similar to those in the previous problem. There exist all color combinations for all different numbers in both orders on the bottom sides with an exception: there do not exist tiles with numbers of connected vertices with the same color. Thus as soon as vertices which are connected and have the same color meet, the assembly stops. Moreover there exist similar tiles with sharp and asterisk with an exception: sharp and the least number, and the largest number and asterisk do not exist because they are used to trigger verification. $\binom{n}{2} \cdot 2 \cdot 9 + 2 \cdot 3(n-1) - 2 \cdot 3e \sim 9n^2 - 6e$ tile types were required.

Border tiles. These tiles are exactly the same like for k-clique. 2 tile types were required.

Verification tiles. These tiles are almost the same like for k-clique, the only difference is that they are triggered by the least and the largest number instead of the least and the largest color, respectively. $3n$ tile types were required.

DONE tile. The tile is exactly the same like for k-clique. 1 tile type was required.

Summed up, the tile complexity of this DNA algorithm is asymptotically equivalent to $9n^2 - 6e$. The binding complexity is asymptotically equivalent to ${}^5/_4\, n^2$, the glue complexity is asymptotically equivalent to ${}^7/_2\, n$.

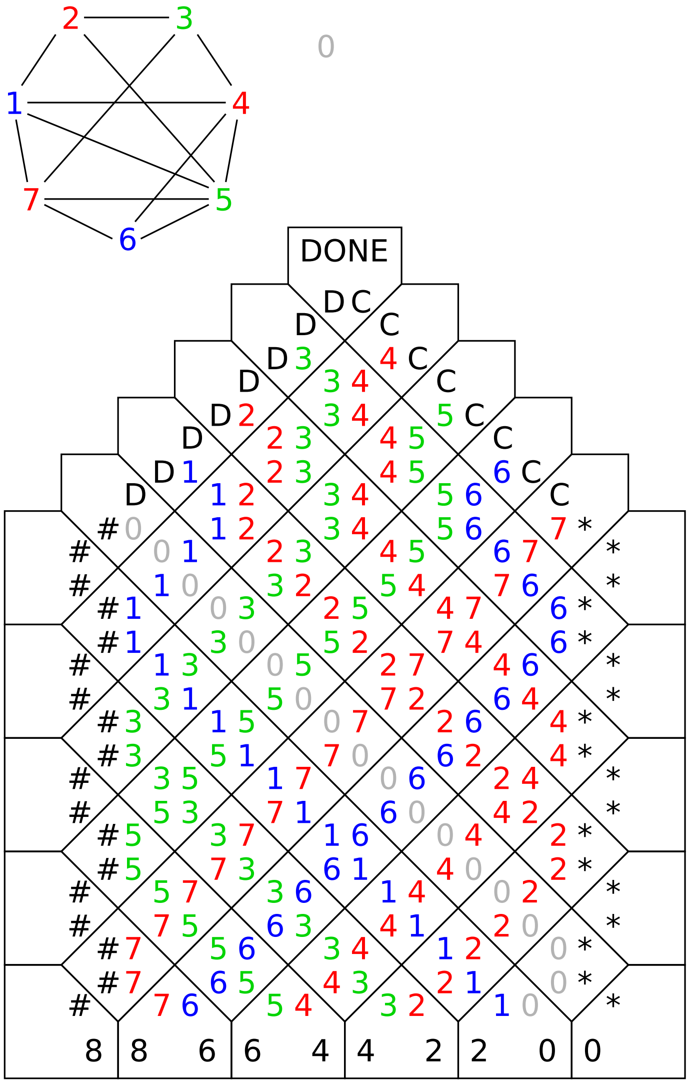


Figure 2.5: 3-color computation.

2.4 Graph Isomorphism Problem

In Example 1.1 we mentioned that Graph Isomorphism Problem is assumed to be neither P nor NP-complete. For this reason it seems to belong to a special class and thus we will describe a DNA system which solves this very special problem.

The following approach is very similar to 3-coloring if we consider n colors instead. The problem can be stated: "For a colorless graph G and a graph H where every vertex has a different color, find a coloring of G with all of those n colors used exactly once (1) such that these colored graphs are isomorphic (2)." Now one has to verify that:

(1) every "color" was used exactly once so that the coloring can define a bijection,

(2) edges and non-edges are preserved.

If k is odd, we add a separated vertex to both graphs resulting in graphs G' and H'. Then G' is isomorphic with H' if and only if G is isomorphic with H. See example in Figure 2.6.

Set of tiles

Bottom tiles. These tiles have almost the same rules as in the graph 3-coloring problem, the difference is that *all* single-color combinations can be omitted. $\frac{n}{2} \cdot n(n-1) \sim \frac{n^3}{2}$ tile types were required.

Bottom corner tiles. These tiles are exactly the same like for the k-clique problem. 2 tile types were required.

Inner tiles. These tiles are similar to those in the graph 3-coloring problem – they are responsible for numerical ordering. The difference is which do exist and which do not. Firstly, there exist only tiles with different numbers and different colors. Now let us assume a tile with numbers k and l with colors a and b, respectively. Note that numbers k and l correspond to vertices in graph G and colors a and b correspond with vertices in graph H. This tile must verify the isomorphism property – existence or non-existence of edge between appropriate vertices. Thus the tile exists if and only if

$$\big(\{k,\, l\} \in E(G) \wedge \{a,\, b\} \in E(H)\big) \vee \big(\{k,\, l\} \notin E(G) \wedge \{a,\, b\} \notin E(H)\big).$$

For similar reasons all pairs of vertices from the graph G meet each other, thus every edge is verified so the condition (2) is satisfied. Moreover it is verified that every color appears exactly once – condition (1): if a color appeared twice with different numbers, these numbers would meet each other and the tiling would stop because a tile with single-color combination does not exist. $4e^2 + 4\big(\binom{n}{2} - e\big)^2 + 2n(n-1) \sim 8e^2 - 4en^2 + n^4$ tile types were required.

Border tiles. These tiles are exactly the same like for the k-clique problem. 2 tile types were required.

Verification tiles. These tiles are almost the same like for the graph 3-coloring problem, the only difference is that there is a different number of color–number combinations. n^2 tile types were required.

DONE tile. The tile is exactly the same like for the k-clique problem. 1 tile type was required.

Summed up, the tile complexity of this DNA algorithm is asymptotically equivalent to $8e^2 - 4en^2 + n^4$. The binding complexity is asymptotically equivalent to $5/4\, n^2$, the glue complexity is asymptotically equivalent to n^2.

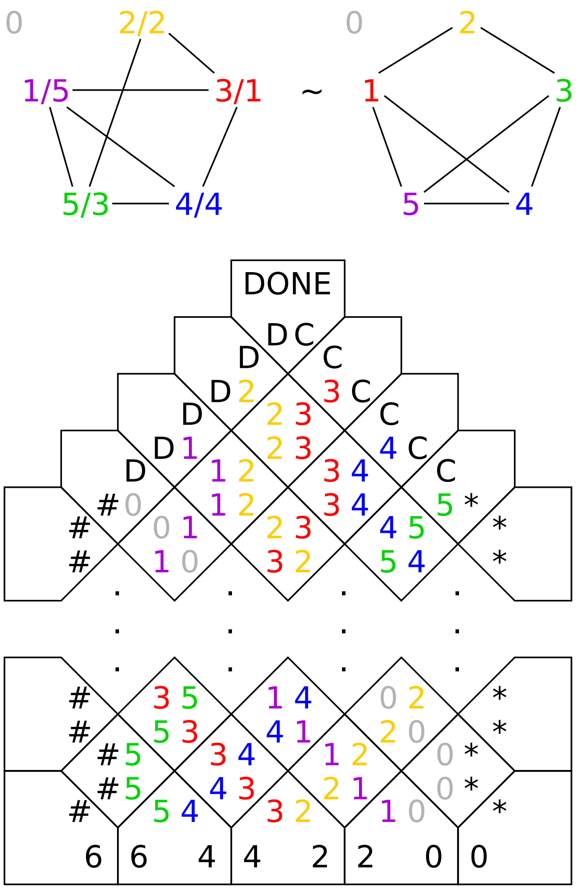


Figure 2.6: Graph isomorphism computation. Color order is defined by their wavelength.

Conclusion

After an introduction to DNA computation, formal languages and complexity theory in Sections 1.1, 1.2, we summarized known results about the relation between strand models and Chomsky hierarchy in Section 1.3.

In Section 1.4 we focused on Wang tile models and presented four kinds of resources for DNA computations. Then we studied the relation between these resources and “classical” resources of Turing machine – time and space. In Section 1.4.4 we presented a new tilesystem which directly simulates a Turing machine at $\tau = 2$ in 2D. We derived upper bounds for all studied complexities with respect to classical resources. We also stated reasonable conditions for a feasible DNA algorithm and as a consequence we concluded that BPP is feasible at $\tau = 2$ in 2D.

In Chapter 2 we derived a model suitable for solving NP problems. Using this model, we presented DNA tilesystems which solve hard computational problems: k-clique Problem, Graph 3-coloring Problem and Graph Isomorphism Problem. These tilesystems were given with all of the studied complexities.

In Section 2.2.1 we described an implementation of our k-clique algorithm in an open source aTAM simulator `xgrow`. The simulation was successful thus it supported correctness of description of our algorithm.

Another interesting problem which arised during the study of Turing universality is $n \times n$ Squares Problem described within Section 1.4.3. This problem might be a direction of future research.

Bibliography

[1] Leonard Adleman, Qi Cheng, Ashish Goel, and Ming-Deh Huang. Running time and program size for self-assembled squares. In *Proceedings of the thirty-third annual ACM symposium on Theory of computing*, pages 740–748. ACM, 2001.

[2] Leonard M Adleman. Molecular computation of solutions to combinatorial problems. *Science - New York then Washington*, pages 1021–1024, 1994.

[3] Leonard M Adleman. On constructing a molecular computer. 1995.

[4] Sanjeev Arora and Boaz Barak. *Computational complexity: a modern approach.* Cambridge University Press, 2009.

[5] Bahar Behsaz, Ján Maňuch, and Ladislav Stacho. Turing universality of step-wise and stage assembly at temperature 1. In *DNA Computing and Molecular Programming*, pages 1–11. Springer, 2012.

[6] Matthew Cook, Yunhui Fu, and Robert Schweller. Temperature 1 self-assembly: Deterministic assembly in 3d and probabilistic assembly in 2d. In *Proceedings of the Twenty-Second Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 570–589. SIAM, 2011.

[7] Richard P Feynman. There's plenty of room at the bottom. *Engineering and Science*, 23(5):22–36, 1960.

[8] Tsu Ju Fu and Nadrian C Seeman. Dna double-crossover molecules. *Biochemistry*, 32(13):3211–3220, 1993.

[9] Godfrey Harold Hardy and John Edensor Littlewood. Some problems of diophantine approximation. *Acta mathematica*, 37(1):155–191, 1914.

[10] Natasha Jonoska and Daria Karpenko. Active tile self-assembly, self-similar structures and recursion. *arXiv preprint arXiv:1211.3085*, 2012.

[11] Donald E Knuth. Big omicron and big omega and big theta. *ACM Sigact News*, 8(2):18–24, 1976.

[12] Paul WK Rothemund and Erik Winfree. The program-size complexity of self-assembled squares. In *Proceedings of the thirty-second annual ACM symposium on Theory of computing*, pages 459–468. ACM, 2000.

[13] Hao Wang. Proving theorems by pattern recognition—ii. *Bell system technical journal*, 40(1):1–41, 1961.

[14] Erik Winfree. On the computational power of dna annealing and ligation. 1996.

[15] Erik Winfree. *Algorithmic self-assembly of DNA*. PhD thesis, California Institute of Technology, 1998.

Appendix A

Contents of attached CD

`/README`	README file,
`/vyzk_ukol.pdf`	Thesis in Portable Document Format,
`/k-clique.rb`	Ruby script which generates an input for a simulation, see Section 2.2.1,
`/graph.yaml`	File with the example graph from Figure 2.3.