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 March 1, 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.,
Konzultant:	—
Abstrakt:	Abstrakt CZ.
Klíčová slova:	Klíčová.

Title:	**Models of self-assembling DNA nanostructures**
Author:	Jakub Klemsa
Abstract:	Abstrakt EN.
Key words:	Keywords.

Contents

1 Introduction to DNA computation **1**

1.1 Basic DNA principles . . . 1

1.2 Complexity, languages . . . 2

1.2.1 O and other notations . . . 2

1.2.2 Studied complexities . . . 2

1.2.3 Languages . . . 3

1.2.4 P, NP and other classes . . . 4

1.3 Strand models . . . 6

1.3.1 Linear strands . . . 6

1.3.2 Adleman's experiment . . . 7

1.3.3 Dendrimer structures . . . 9

1.3.4 Double crossover molecules . . . 9

1.4 Wang-tile models . . . 10

1.4.1 Definition . . . 10

1.4.2 Computational power . . . 10

2 Solutions to NP problems **13**

2.1 Abstract model for DAE units . . . 13

2.1.1 Feasibility of the class BPP . . . 13

2.2 Graph 3-coloring . . . 14

2.3 Graph isomorphism . . . 17

2.4 k-clique . . . 20

References **22**

Chapter 1

Introduction to DNA computation

1959: Feynman's visionary talk, [7]; pros: extreme paralelism, cons: reliability.

The ground-breaking work was carried out by Adleman, [2], who showed that DNA computation is practically feasible. In his experiment, Adleman used special DNA sequences for solving Hamiltonian Path Problem, one of the most typical NP-complete problems.

... Extreme parallelism! But also possibility of errors.

Work overview

Chapter 1: Intro.
Chapter 2: First of all we will describe models which exploit specific DNA structure.
Chapter 3: Abstract Tile Assembly Model, temperature, 2D vs. 3D.
Positive integers $\mathbb{N}$.

1.1 Basic DNA principles

DNA (deoxyribonucleic acid) is a large biomolecule carrying living organisms' genetic information. Its most common structure is 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 nucleobases in DNA (+1 in RNA): Adenine (**A**), Thymine (**T**), Cytosine (**C**), Guanine (**G**) and Uracil (**U**) in RNA instead of Thymine in DNA. These molecules are responsible for making hydrogen bonds between strands in a manner following Watson-Crick complementarity: only **A** – **T** and **C** – **G** pairs can be formed.

Backbone molecules Backbone of DNA strand is made of alternating deoxyriboses and phosphates. Phosphates only hold adjacent deoxyriboses, each deoxyribose moreover holds one nucleobase. Due to 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, languages

1.2.1 O and other notations

Let us briefly remind O-, o-, Ω-, ω- and Θ-notations for $f, g : \mathbb{N} \to \mathbb{N}$. 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 infinitedly many n".

Definition 1.1.

$$g \in O(f) \Leftrightarrow (\exists C > 0)(\forall^* n)\big(g(n) < Cf(n)\big)$$

$$g \in \omega^{(1)}(f) \Leftrightarrow (\forall C > 0)(\exists^\infty n)\big(g(n) > Cf(n)\big)$$

$$g \in \omega^{(2)}(f) \Leftrightarrow (\forall C > 0)(\forall^* n)\big(g(n) > Cf(n)\big)$$

$$g \in \Omega^{(1)}(f) \Leftrightarrow (\exists C > 0)(\exists^\infty n)\big(g(n) > Cf(n)\big)$$

$$g \in \Omega^{(2)}(f) \Leftrightarrow (\exists C > 0)(\forall^* n)\big(g(n) > Cf(n)\big)$$

$$g \in o(f) \Leftrightarrow (\forall C > 0)(\forall^* n)\big(g(n) < Cf(n)\big)$$

$$g \in \Theta(f) \Leftrightarrow (\exists C_1, C_2 > 0)(\forall^* n)\big(C_1 f(n) \le g(n) \le C_2 f(n)\big)$$

$$g \sim f \Leftrightarrow \lim_{n\to\infty} \frac{g(n)}{f(n)} = 1$$

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

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

The other, $\Omega^{(2)}$, was introduced by Knuth from good reasons described in [10]. In similar manner there are two definitions for ω.

Note that there are also some relations: the condition for $\Omega^{(1)}$ is negation of the condition for o so these sets are complementary for 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:

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

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 Studied complexities

Definition 1.2. All the following complexities are considered as functions of problem size n:

Biostep complexity will refer to the number of laboratory steps required to handle the computation, denoted by $Bs(n)$.

Binding complexity will refer to the number of bindings in given computation, denoted by $Bnd(n)$.

Tile complexity will refer to the number of different DNA tiles, denoted by $Ti(n)$.

Glue complexity will refer 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.

Remark 1.2. Note some properties of proposed complexities.

Ad biostep complexity Adleman [3] describes formally in his *unrestricted model* few types of such lab procedures – *Separate, Merge, Detect* and *Amplify*, Winfree [14] adds another – *Append*. And 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.

Ad binding complexity Too high binding complexity leads to lower probability of correct computation P_c because it holds $P_c(n) = (1 - p_e)^{Bnd(n)} \approx^{1} 1 - p_e \cdot Bnd(n)$ where p_e denotes probability of erroneous binding.

Ad tile complexity The higher tile complexity the more demanding it is to prepare required tiles.

Ad glue complexity Higher glue complexity will require longer DNA sequences in the sticky ends.

1.2.3 Languages

Definition 1.3. Let Σ be an nonempty and finite set of *characters* which will be referred to as *alphabet*. Define set of *words* over alphabet Σ as $\Sigma^* = \bigcup_{n \in \mathbb{N}_0} \Sigma^n$ and set of nonempty words as $\Sigma^+ = \bigcup_{n \in \mathbb{N}} \Sigma^n$. Empty word will be denoted by ε. *Language* $\mathcal{L}$ over alphabet Σ is then just a set of words: $\mathcal{L} \subseteq \Sigma^*$. *Complement* of 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. Denote alphabet of non-terminals by N, alphabet of terminals by Σ and initial symbol by I.

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 these rules without those having I on the right hand side, then there is allowed also the rule $I \to \varepsilon$.

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 $A \to bB$ and $A \to b$ where A, $B \in N$ and $b \in \Sigma$. Or, if we assume these rules without those having I on the right hand side, then there is allowed also the rule $I \to \varepsilon$.

[1] If reasonable.

Remark 1.3. 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 $g(n) = 1 \Leftrightarrow f(n) \in \mathcal{L}$.

1.2.4 P, NP and other classes

This section describes few classes of languages from the resource-consumption point of view. There exist many equivalent definitions, these are taken from [4].

Definition 1.4. $\mathcal{L} \subseteq \Sigma^*$. $\mathcal{L} \in \mathsf{P} \Leftrightarrow \big(\exists p \in \mathcal{P}\big)\big(\exists$ deterministic Turing machine $M\big)$ $\big(\forall x \in \Sigma^*\big)\big(x \in \mathcal{L} \Leftrightarrow M$ accepts x in time $\leq p(|x|)\big)$.

Definition 1.5. $\mathcal{L} \subseteq \Sigma^*$. $\mathcal{L} \in \mathsf{NP} \Leftrightarrow \big(\exists p, q \in \mathcal{P}\big)\big(\exists$ deterministic Turing machine $M\big)$ $\big(\forall x \in \Sigma^*\big)\Big(x \in \mathcal{L} \Leftrightarrow \big(\exists y \in \Sigma^{p(|x|)}\big)\big(M$ accepts (x, y) in time $\leq q(|x| + |y|)\big)\Big)$.
Such y will be referred to as *certificate* for x (with respect to $\mathcal{L}$ and M).

Definition 1.6. $\mathcal{L} \subseteq \Sigma^*$. $\mathcal{L} \in \mathsf{co\text{-}NP} \Leftrightarrow \big(\exists p, q \in \mathcal{P}\big)\big(\exists$ deterministic Turing machine $M\big)$ $\big(\forall x \in \Sigma^*\big)\Big(x \in \mathcal{L} \Leftrightarrow \big(\forall y \in \Sigma^{p(|x|)}\big)\big(M$ accepts (x, y) in time $\leq q(|x| + |y|)\big)\Big)$.

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 ambiguous transition function,
- it has two types of terminal states: accepting q_{accept} and halting without accepting q_{halt} (not rejecting!),
- we say that it accepts input x iff there *exists* a sequence of decisions ending in q_{accept}, as consequence it declines x iff *for every* sequence of decisions reaches q_{halt},

Definition 1.7. $\mathsf{DTime}\big(f(n)\big) = \big\{\mathcal{L}$ language $\big|$ $(\exists$ deterministic Turing machine $M)$ $(\forall x \in \Sigma^*)\big(x \in \mathcal{L} \Leftrightarrow M$ accepts x in time $\leq f(|x|)\big)\big\}$.
Other classes are defined in a very similar manner: time $\leftrightarrow$ space, deterministic $\leftrightarrow$ non-deterministic.

Definition 1.8. $\mathsf{P} = \bigcup\limits_{k \in \mathbb{N}} \mathsf{DTime}(n^k)$ and $\mathsf{NP} = \bigcup\limits_{k \in \mathbb{N}} \mathsf{NTime}(n^k)$.

Definition 1.9. $\mathcal{L} \in \mathsf{co\text{-}NP} \Leftrightarrow \overline{\mathcal{L}} \in \mathsf{NP}$.

Note 1.4. $\mathsf{NP} \subseteq \mathcal{P}(\Sigma^*)$ so the notation $\mathsf{co\text{-}NP}$ might be confusing. Note that $\mathsf{co\text{-}NP}$ is *not* a complement to NP as a set of languages.

Remark 1.5. 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, non-deterministic is not known to. Note that Immerman–Szelepcsényi theorem states possibility of non-deterministic Turing machine negation in limited *space*: $\mathcal{L} \in \mathsf{NSpace}(s(n)) \Leftrightarrow \overline{\mathcal{L}} \in \mathsf{NSpace}(s(n))$ for $s(n) \geq \log(n)$.

Theorem 1.1. All the previous definitions of P, NP and $\mathsf{co\text{-}NP}$ are equivalent.

Definition 1.10. We say that language $\mathcal{L}_1$ is *polynomial-time Karp reducible* to $\mathcal{L}_2$ and denote $\mathcal{L}_1 \propto \mathcal{L}_2 \Leftrightarrow \exists$ polynomial-time computable function $f : \Sigma^* \to \Sigma^*$ such that $(\forall x \in \Sigma^*)(x \in \mathcal{L}_1 \Leftrightarrow x \in \mathcal{L}_2)$.

Definition 1.11. $\mathcal{L} \in \mathsf{NP}\text{-hard} \Leftrightarrow (\forall \mathcal{L}' \in \mathsf{NP})(\mathcal{L}' \propto \mathcal{L})$.

Definition 1.12. $\mathcal{L} \in \mathsf{NP}\text{-complete} \Leftrightarrow \mathcal{L} \in \mathsf{NP} \cap \mathsf{NP}\text{-hard}$.

Note 1.6. 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 supposed that $\mathsf{P} \neq \mathsf{NP}$ there does not exist a polynomial algorithm for any NP-complete language.

Example 1.1. Remind some popular NP-complete problems: boolean formula satisfability (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 few interesting problems which are not known to belong either to P nor to NP-complete, one of them is graph isomorphism problem.

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

We say that PTM M

- decides language $\mathcal{L}$ iff for every $x \in \Sigma^*$ the probability of halting in correct state (i.e. q_{accept} for $x \in \mathcal{L}$ and q_{reject} for $x \notin \mathcal{L}$) is higher than $2/3$.
- decides language $\mathcal{L}$ in time $t(n)$ iff M decides language $\mathcal{L}$ and for every $x \in \Sigma^*$ it halts in time $\leq t(|x|)$.

Now we can define those classes, note that it would be possible to use again two types of definition like above.

Definition 1.13. $\mathsf{BPTime}\big(f(n)\big) = \big\{\mathcal{L} \text{ language} \,\big|\, (\exists \text{ probabilistic Turing machine } M)$ $\big(M \text{ decides } \mathcal{L} \text{ in time } \leq f(|x|)\big)\big\}$.

Definition 1.14. $\mathsf{BPP} = \bigcup\limits_{k \in \mathbb{N}} \mathsf{BPTime}(n^k)$.

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

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

Definition 1.17. $\mathcal{L} \in \mathsf{co\text{-}RP} \Leftrightarrow \overline{\mathcal{L}} \in \mathsf{RP}$.

Remark 1.7. 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 error for $x \notin \mathcal{L}$ and $\mathsf{co\text{-}RP}$ does not allow error for $x \in \mathcal{L}$. Following class ZPP (from *zero error*) does not allow any error, it only allows halting neither with accepting nor with rejecting.

Definition 1.18. $\mathsf{ZPTime}\big(f(n)\big) = \Big\{\mathcal{L} \text{ language} \,\Big|\, \big(\exists \text{ probabilistic Turing machine } M\big)$
$\big(\forall x \in \Sigma^*\big)\Big(M \text{ halts in time } \leq f(|x|) \ \wedge$
$\big(x \in \mathcal{L} \Rightarrow \mathbb{P}(M \text{ accepts } x) \geq 2/3 \wedge \mathbb{P}(M \text{ rejects } x) = 0\big) \ \wedge$
$\big(x \notin \mathcal{L} \Rightarrow \mathbb{P}(M \text{ rejects } x) \geq 2/3 \wedge \mathbb{P}(M \text{ accepts } x) = 0\big)\Big)\Big\}$.

Definition 1.19. $\mathsf{ZPP} = \bigcup_{k \in \mathbb{N}} \mathsf{ZPTime}(n^k)$.

Theorem 1.2.

$$\mathcal{L}_{1,2} \in \mathsf{ZPP} \Rightarrow \overline{\mathcal{L}}_1,\ \mathcal{L}_1 \cap \mathcal{L}_2,\ \mathcal{L}_1 \cup \mathcal{L}_2 \in \mathsf{ZPP}.$$

Theorem 1.3.

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

1.3 Strand models

There exist (can be synthesized) many types of molecules, well described in 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 1.8. DNA naturally forms double-helices which would be confusing in figures so there will mostly appear schemes of "untwisted" molecules, double crossover molecules will be described more precisely.

1.3.1 Linear strands

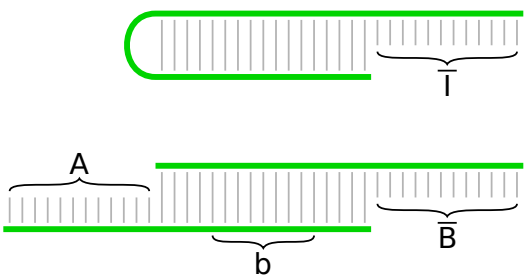


Figure 1.1: Linear strands: Strand for initial symbol I and strand for rule $A \to bB$.

Equivalent to regular languages: Winfree pg 53.

1.3.2 Adleman's experiment

Adleman showed in his ground-breaking work [2] that DNA molecules are really capable of computation. He exploited that huge parallelism possible in DNA computation for one of the most fundamental NP-complete problems – the Hamiltonian Path Problem (HPP) in 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.

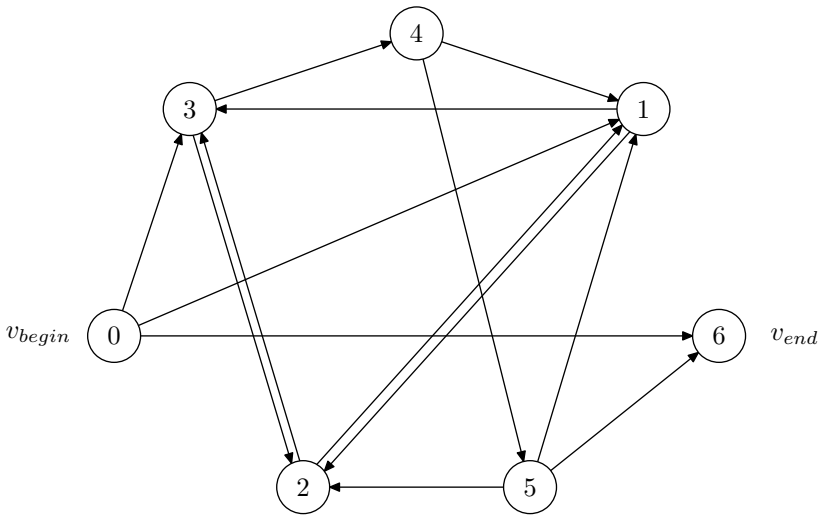


Figure 1.2: Adleman's original graph.

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[2].

Adleman first presents this non-deterministic five-step algorithm, whose steps are then described 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 paths remain, say "Yes"; otherwise, say "No."[3]

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

[2]Note that it can be re-labelled such a nice way without loss of generality.

[3]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 1.2.4.

associated with a random[4] 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 $\overline{q_i p_j}$ sequence with reverse backbone orientation ($3' \to 5'$) where $\overline{q_i}$ stands for Watson-Crick complementary word. There is an exception for $i = begin$ and $j = end$: instead of $\overline{q_{begin} p_j}$ there is $\overline{O_{begin} p_j}$ and in a similar way for $j = end$.

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

Figure 1.3: Example of assigned sequences.

It can be easily seen that all correctly bonded double-strands correspond with a valid walk through G_n. Moreover, all complete double-strands 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 the product of Step 3 with a *biotin-avidin magnetic beads system*. This process extracts consequently for every vertex i only those DNA strands which contain a substring representing vertex i. Thus this algorithm has biostep complexity $O(n)$ which we considered unfeasible. Better solution with biostep complexity $O(1)$ was introduced by Winfree [14], see section 1.3.4.

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

1.3.3 Dendrimer structures

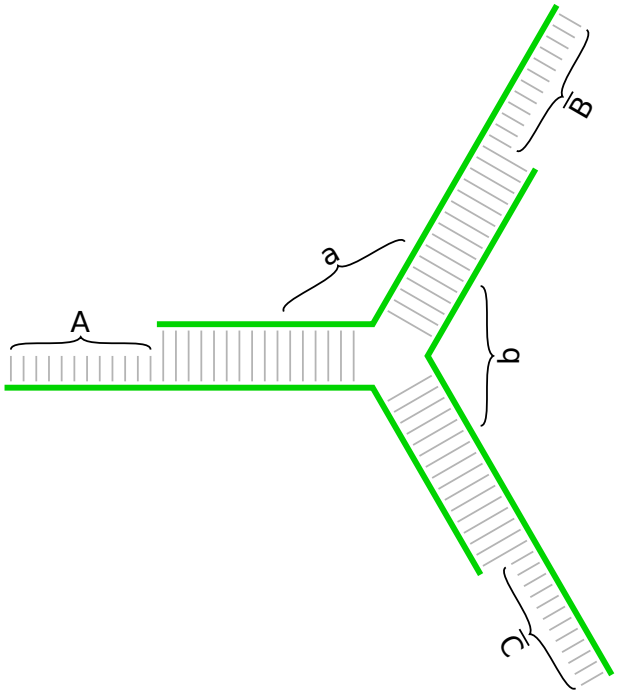


Figure 1.4: Dendrimer structure for rule $A \to aBbC$.

Equivalent to context-free languages: Winfree pg 54.

1.3.4 Double crossover molecules

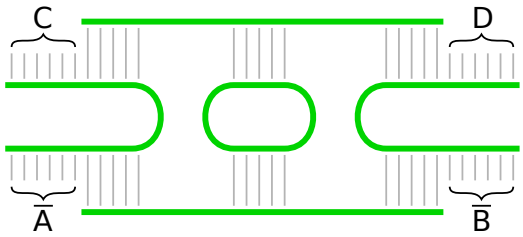
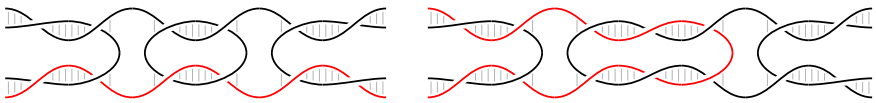


Figure 1.5: Double crossover molecule scheme.

Double crossover molecules are the most important ones. Though they are more complicated they are still very rigid [8]. Moreover they are theoretically capable of universal computation because they form 2D tilings and thus they can simulate blocked cellular automaton [14]. As we mentioned in note 1.8 we will describe also their inner structure.

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 consisting of small parst from all stages in the tiling, on the other hand DAE molecules form tilings with a strand leading through entire stage in a row. Later we will need to read the bottom line's sequence thus it is practically reasonable to use DAE molecules. Moreover we will require even number of half-turns between crossovers in adjacent molecules so that the bottom line can be read as is, with no Watson-Crick complementarities.



(a) DAE. (b) DAO.

Figure 1.6: DAO vs. DAE.

DAE molecules seem to be the most promising thus in following text we will be interested only in these molecules.

1.4 Wang-tile models

1.4.1 Definition

More abstract model where one handles only with "glues" on edges of Wang tiles. Define *temperature*.

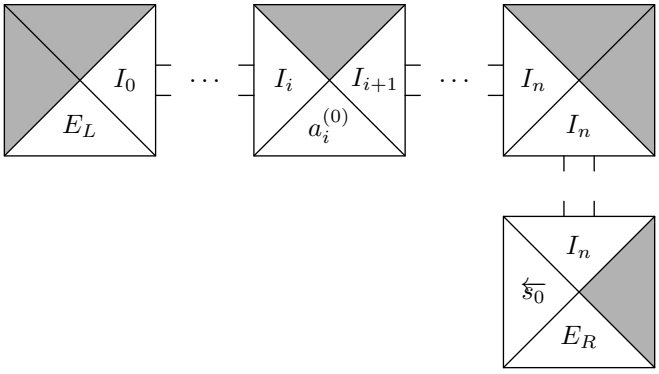
1.4.2 Computational power

Give table of Turing universality [6].

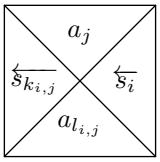
See [14]. Many other results in [6], [5], [13], [1] ...

Turing universality of 2D tiles at $T = 2$

Input tape:



Comes from right, continues left:



Comes from right, continues back to the right:

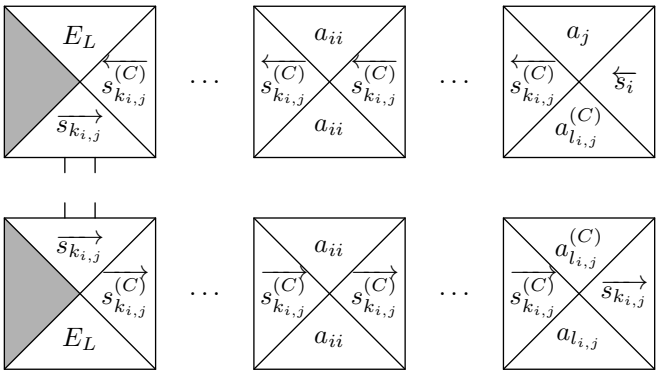
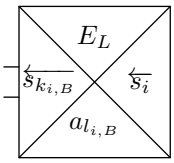
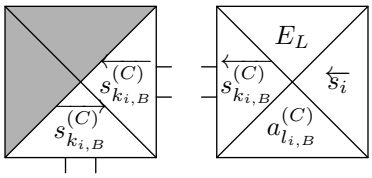


Figure 1.7: Tileset 1/2.

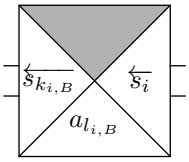
Reads EndLeft, continues left:



Reads EndLeft, continues back to the right:



Comes from right and reads Blank, continues left:



Comes from right and reads Blank, continues back to the right:

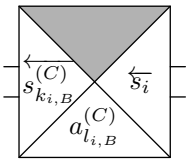


Figure 1.8: Tileset 2/2.

error-tolerant rules – Gacs and Reif, 1988

Chapter 2

Solutions to NP problems

2.1 Abstract model for DAE units

In this section we will introduce a simplifying model for Winfree's solution to the HPP. We have two reasons:

1. the model will be based on the aTAM model,
2. pictures will be easier-to-read.

Definition 2.1. Let us define aaTAM (augmented aTAM) as aTAM with one reduction and one augmentation:

- aTAM is reducted so that it only allows given glue-strengths for given tile-types defined in next point,
- augmented by 4 other tile types from figure.

Remark 2.1. Those tiles with glue-strength 2 will be used for generating random words. Note that these words can be restricted by arbitrary regular grammar rule.

Note those twice longer sticky ends on the bottom line.

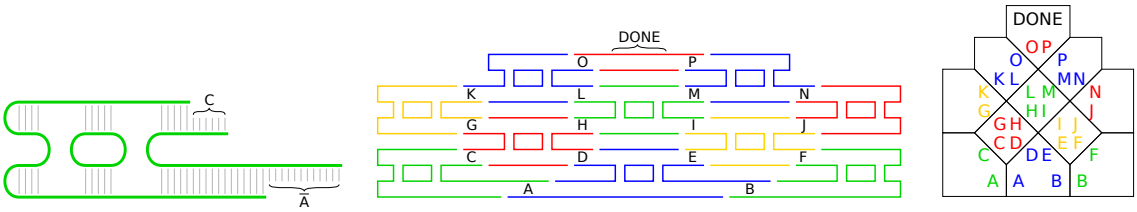
In following examples this model will be set up to act similarly like NP: $\exists y \; R(x, y)$. Although existence is not sure, it is very likely. Predicate R will be "enumerable in polynomial time" for $x \in L$. In this context, *enumerable in polynomial time* will mean number of bindings, not number of biosteps. This can be assumed due to Turing universality of this model in $O(1)$ biosteps – biostep complexity is not restrictive[1] and will be required to be $O(1)$ due to its lab complexity. On the other hand the binding complexity will be very important, we will be interested even in constants. This is because the less binding complexity, the less probability of error.

Note that the word on the bottom line can be restricted to belong to arbitrary regular language.

2.1.1 Feasibility of the class BPP

As it is often conjectured that the class P is the class of somehow practically feasible problems, here we introduce similar conjecture for feasible DNA algorithms. The corresponding debate can be found in [4].

[1] From Turing's thesis, Turing machine is the most universal model.



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

Figure 2.1: Evolution of aaTAM model from DAE units to tiles. Here glues A and B are of strength 2.

Conjecture 2.1. Feasible DNA algorithms comply $Bs(n) \in O(1)$; $Bnd(n)$, $Ti(n)$, $Gl(n) \in \mathsf{P}$.

Theorem 2.2. Probabilistic Turing machine can be simulated by probabilistic cellular automaton.

Proof. Can be found in ... □

Theorem 2.3. "Abstract model for DAE units" can simulate probabilistic cellular automaton.

Proof. Blabla. □

Corollary 2.4. BPP is decidable by abstract model.

2.2 Graph 3-coloring

Remind original Knuth's algorithm at `http://www.iti.fh-flensburg.de/lang/algorithmen/sortieren/networks/oetsen.htm`! And prove that everything goes fine! Robusticity?

First idea: Generate all bonds with colored atoms and check the entire system (haha, complexity like $O(n^4)$ because $|E| \in O(n^2)$). Second solution: Generate a reverse-order sequence of vertices and let it order in the correct order. All pairs should meet each other, the problem to solve is whether all pairs really meet each other. After that check that the area is full like Winfree – from one side to the other. Improvement: the check can be triggered from both sides simultaneously.

Set of tiles

First of all the graph needs to have even number of vertices, thus one separated non-colored vertex has to be added if applicable. Then follow these rules which are showed in an example, see figure 2.3.

Bottom line For every pair $(2k, 2k+1)$ there will be a bottom-type tile with non-colored numbers $(2k+2, 2k)$ on the bottom and with all feasible[2] color combinations of $(2k+1, 2k)$ on the top. From practical decoding reasons (see Winfree [14]) the sequences encoding colored numbers on the top must be physically present also wherever on the bottom DNA strand, see figure 2.2. $\frac{9n}{2}$ tile types were required.

[2] If $(2k+1, 2k)$ are connected, same-colored numbers are omitted.

Bottom corner tiles Both corner tiles are connected on the bottom by the highest and the lowest non-colored number, respectively, and have their special glue (# for $-\infty$ and * for $+\infty$, respectively) on the top. These first two sets of tiles generate all colorings of given graph (without those omitted in previous step) with reverse order of numbers. 2 tile types were required.

Inner tiles These tiles are responsible for ordering[3]. There exist all color combinations for all different numbers with two important exceptions. There *do not* exist tiles with numbers of connected vertices with the same color. Thus, as soon as there appears such forbidden combination, the self assembly cannot continue and reach "DONE". Because the numbers are generated in reverse order they must meet each other – note that they simply cannot "jump" and every number has to exchange with all the higher ones as well as with the lower ones. This implies that every forbidden combination would be revealed, thus it answers correctly if and only if the coloring is correct. The second exception are those described in the following paragraph. $9n^2$ tile types were required.

Border tiles There are two tile types on the borders, one with sharp, one with asterisk. They keep the structure growing up.

Checking tiles As soon as the biggest and the smallest number reach * and #, respectively, there are two special tiles which start checking whether nothing is missing. Note that all tiles had time enough to get into correct order. In this setup checking tiles do not need to check correct order thus there can exist only two types of checking sequences "C" and "D" with all color-number combinations of middle numbers – "D" with the smaller half, "C" with the higher half. $3n$ tile types were required.

DONE tile If everything is checked and checking sequences meet each other, "DONE" tile will be connected to signalize correct solution. 1 tile type was required.

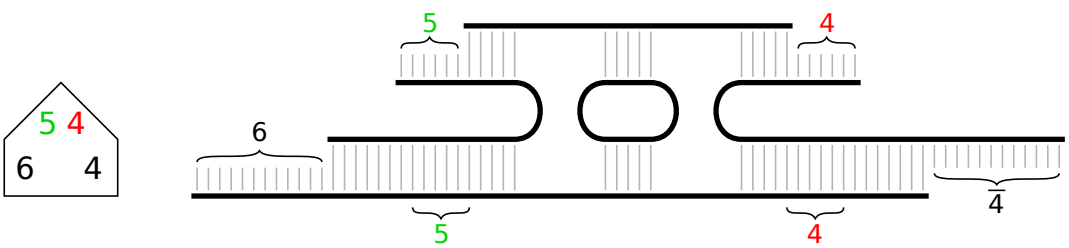


Figure 2.2: Bottom tile with desired sequences in the bottom strand.

Summed up, this DNA algorithm requires $9n^2$ tile types. Glue complexity is ...

The first idea's binding complexity was like $O(n^4)$, the second is already $O(n^2)$, the binding complexity is $1^1/_2\, n^2$. The improvement decreases it to $1^1/_4\, n^2$.

[3]Principially they are the same as in Winfree [14].

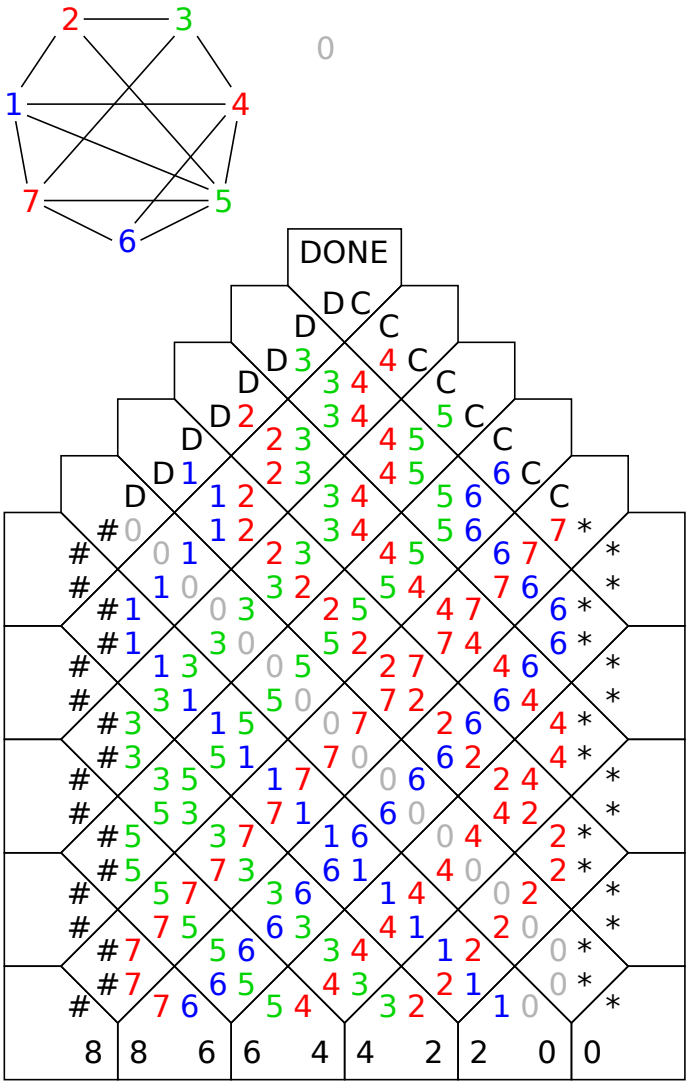


Figure 2.3: 3-color computation.

2.3 Graph isomorphism

In example 1.1 we mentioned that graph isomorphism problem seems to be neither P nor NP-complete. From this reason it seems to belong to a special class and thus we will describe a DNA system which solves this very special problem.

Following approach is very similar to 3-coloring if we consider n colors instead. The problem can be stated: "For a non-colored graph G and a graph H where every vertex is colored with 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 check that:

1. every "color" was used exactly once so that it is a bijection,
2. edges and non-edges are preserved.

Set of tiles

Like before, the graph needs to have even number of vertices, thus one separated self-bijective vertex has to be added if applicable. An example is given, see figure 2.5.

Bottom line These tiles have almost exactly the same rules as in graph 3-coloring, the difference is that *all* of the same-colored combinations are omitted. $\frac{n^3}{2}$ tile types were required.

Bottom corner tiles Are exactly the same. 2 tile types were required.

Inner tiles There are three types of inner tiles:

Number-ordering tiles These are similar to previous ones, the difference is which do exist and which do not. Let us assume a tile with numbers k and l with colors a and b, respectively. Note that numbers k and l correspond with vertices in graph G and colors a and b correspond with vertices in graph H. This tile must check the isomorphism property – existence or non-existence of edge between appropriate vertices. Thus the tile exists if and only if

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

From similar reasons all pairs of vertices from graph G meet each other, thus every edge is checked so condition number 2 would be done. n^4 tile types were required.

Color extracting tiles Now we have to extract colors (forget numbers) and order them in given order so that we can check that every color is used exactly once. This process will be triggered by a special inner tile with the highest number of arbitrary color and a non-colored asterisk on the bottom. On the top it will have an asterisk of that number's color and a non-colored asterisk. For every other number with arbitrary color there exists a tile with it and an asterisk of an arbitrary but different color on the bottom. On the top it will have two asterisks of these colors in correct order. n^3 tile types were required.

Color-ordering tiles These are similar to those with numbers. Similarly there do not exist tiles with one color. n^2 tile types were required.

Border tiles These tiles are exactly the same like for 3-coloring.

Checking tiles As soon as there appears a combination of sharp and most-left-colored asterisk, a checking tile comes having "C" of the second color on the right top. After this initialization there are tiles with colored "C" and same-colored asterisk on the bottom and next-colored "C" on the right top. This ensures that every color was used exactly once. The last color is followed by non-colored "C". n tile types were required.

DONE tile Finally if non-colored "C" meets non-colored asterisk, a "DONE" tile is connected signalizing correct solution. 1 tile type was required.

Summed up, this DNA algorithm requires n^4 tile types and its binding complexity is $2^1/_2\, n^2$. Glue complexity is ...

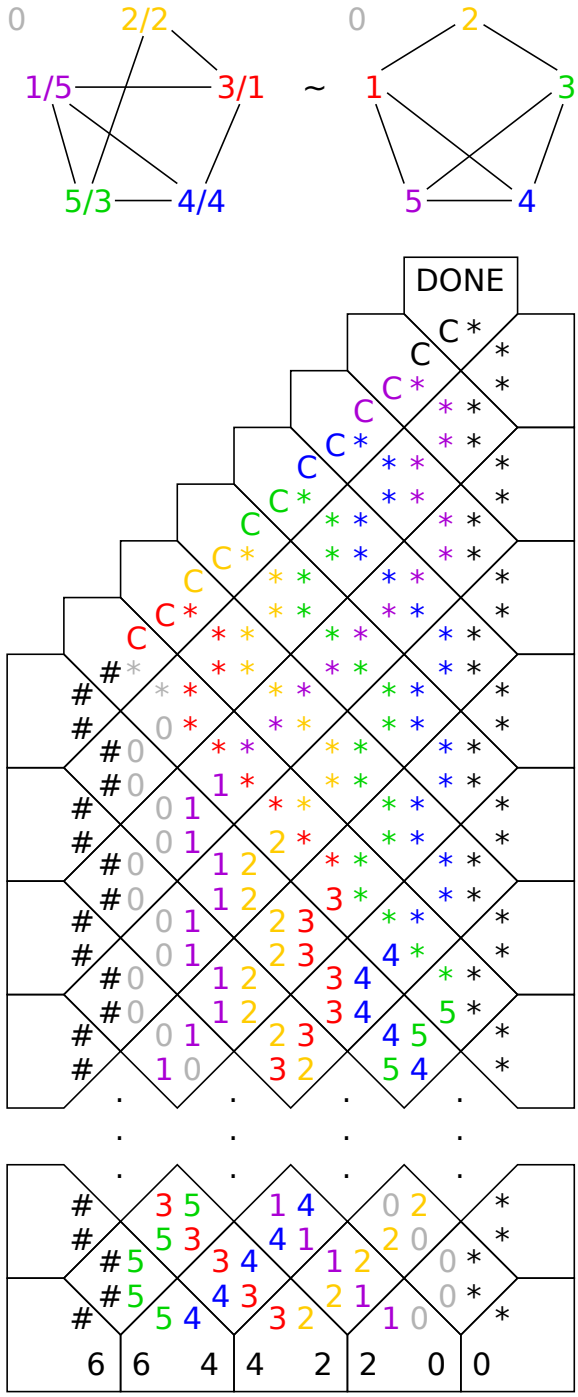


Figure 2.4: Graph isomorphism computation. Color order is defined by their wavelength.

2.4 k-clique

k-clique problem belongs to NP-complete problems. Note that k-clique problem in G is equivalent to k-independent set in $\overline{G}$ and that is equivalent to $n-k$-vertex cover of $\overline{G}$ so we can assume $k \leq \frac{n}{2}$. Like before we add an unchecked vertex if k is odd so we will assume k to be even.

Set of tiles

Bottom line For now there are tiles with $2l-2$ and $2l$ $(0 < l \leq \frac{k}{2})$ on the bottom and with an arbitraty ordered[4] pair of different numbers from 1 to n with $k-2l+2$-th and $k-2l+1$-th colors, respectively, on the top. Note that now the order of colors is given and do not forget that they should also contain those upper sequences once more on the bottom strand. $\frac{kn^2}{4}$ tile types were required.

Bottom corner tiles Are exactly the same. 2 tile types were required.

Inner tiles These tiles are now responsible for ordering by color during which they check existence of *every* edge in similar manner to previous problems. And because the first line contains them in reverse order there will meet each other. k^2n^2 tile types were required.

Border tiles These tiles are exactly the same like for 3-coloring.

Checking tiles As soon as the most left color reaches sharp and the most right color reaches asterisk, checking is triggered in similar manner to 3-coloring. kn tile types were required.

DONE tile This is exactly the same like 3-coloring. 1 tile type was required.

Summed up, this DNA algorithm requires k^2n^2 tile types. Binding complexity is $1^1/_4$ $\min\{k^2, (n-k)^2\}$ Glue complexity is ...

[4]Note that this restriction does not reduce the set of possible k-member subsets.

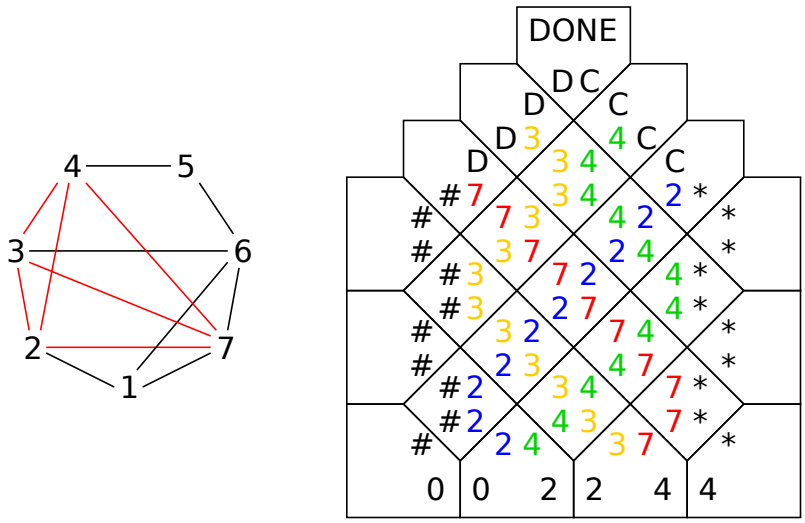


Figure 2.5: k-clique computation. Color order is defined by their wavelength.

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

[11] Richard J Lipton. Speeding up computations via molecular biology. *DNA Based Computers*, 27:67–74, 1996.

[12] Richard J Lipton et al. Dna solution of hard computational problems. *Science*, 268(5210):542–545, 1995.

[13] 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.

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