Overview and code simulation of DNA computing to solve Boolean satisfiability problem

Hongjun Jiang[1]

[1]Southern University of Science and Technology, Shenzhen, Guangdong, China
E-mail: 12411912@mail.sustech.edu.cn

1 Abstract

NP-complete problems and the Boolean Satisfiability Problem (SAT) are core issues in logic and computer science, whose solutions can significantly impact the efficiency of solving various optimization problems. Consequently, they have consistently been the focus of scholarly attention. DNA computing is a novel paradigm that primarily leverages the massive parallelism, high-density storage capacity, and programmable hybridization/dissociation operations of biological molecules like DNA to perform computations, providing an excellent experimental platform for SAT problems. This paper first reviews NP problems, SAT/3-SAT problems, and the Cook-Levin reduction approach; then it primarily introduces the core principles of DNA computing, selecting two representative examples of DNA computing solving SAT problems: *Qinghua Liu (2000)*[1] and *Ravinderjit S. Braich et al. (2002)*[2], analyzing their principles, details, and subsequent technological development paths; subsequently, it combines Python simulation code to analyze the steps of encoding, screening, amplification, and readout in DNA computing, using code snippets to demonstrate the digital implementation of biochemical abstractions such as "amplification-screening-ligation-electrophoresis-readout," and adds visualizations to display the DNA evolution process; finally, it discusses the potential future development paths of DNA computing in larger-scale SAT problems and other complex problems from the perspectives of error, complexity, programmability, and future directions.

Keywords: **DNA computing**, **SAT problem**, **parallel computing**, **biochemical reaction simulation**, **biochemical simulator**

2 Introduction

The Boolean Satisfiability Problem (SAT) is a core issue in logic and computer science, aiming to determine whether there exists a set of variable assignments that makes a given conjunctive normal form (CNF) Boolean formula evaluate to true. It is also the first problem proven to be NP-complete, with a polynomial-time reduction from any $L \in \mathrm{NP}$ to SAT ($L \leq_P \mathrm{SAT}$), meaning that any NP problem can be transformed into a SAT problem in polynomial time. Consequently, its solution efficiency directly impacts many combinatorial optimization problems. 3-SAT restricts each clause to contain at most three literals while still maintaining NP-completeness. Classical solvers (DPLL/CDCL, local search, Survey Propagation) are efficient on many instances but still face exponential time in worst-case scenarios (phase transition regions): the worst-case upper bound is known to be $O^*(1.439^n)$ and the lower bound $\Omega(1.3^n)$. DNA computing, characterized by molecular parallelism, can cover an extremely large number of candidate solutions simultaneously, completing the generation-screening-amplification-readout process through hybridization selection, enzymatic cleavage, PCR amplification, separation, and readout, providing an efficient parallel path for NP problems. The experiment conducted by *Liu* in 2000 was the first demonstration of SAT computation on a solid surface (gold film), solving a 3-SAT problem with 4 variables and 4 clauses. *Ravinderjit S. Braich et al.* in 2002 adopted the "sticker model" and experimentally solved a then unprecedented 3-SAT problem with 20 variables and 24 clauses. Although DNA computing faces numerous practical limitations, such as slow electrophoresis rates and complex pre-preparation requirements, its parallelism concept and the encoding paradigm based on base complementarity have inspired many computer algorithms, such as graph neural networks (GNN) and DNA genetic algorithms. The objective of this paper is to link SAT and DNA computing at both mathematical and experimental scales, providing reproducible Python simulator examples and visualizations of the DNA evolution process, thereby connecting formulas, biochemical operations, and code interfaces.

3 Background Theory

3.1 Theoretical Basis of NP and SAT Problem

NP and NP-Completeness

NP problems refer to a class of decision problems whose solutions can be verified in polynomial time. A language $L \subseteq \{0,1\}^*$ belongs to the NP class if and only if there exists a polynomial p and a polynomial-time verifier (Turing machine) V such that:

$$x \in L \Leftrightarrow \exists y,\ |y| \leq p(|x|),\ V(x,y) = 1$$

For a problem A to be NP-complete, it must satisfy: (1) $A \in \mathrm{NP}$; and (2) for any problem $B \in \mathrm{NP}$, there exists a polynomial-time reduction, i.e.:

$$(1)A \in \mathrm{NP};\ \ (2)\forall B \in \mathrm{NP},\ B \leq_P A_\circ$$

NP-complete problems are those in the NP class that satisfy the property that "any NP problem can be reduced to this problem in polynomial time." These problems have significant theoretical importance, as finding a polynomial-time algorithm for any NP-complete problem could potentially lead to the important conclusion that P=NP.

SAT Definition and Cook-Levin Reduction

The Boolean Satisfiability Problem (SAT) is defined as follows:
Given a Boolean formula $\phi(x_1, x_2, \ldots, x_n)$ in conjunctive normal form (CNF):

$$\phi(x_1, \ldots, x_n) = \bigwedge_{i=1}^{m} C_i, \quad C_i = \bigvee_{j \in S_i} l_j,\ l_j \in \{x_k, \neg x_k\}$$

Determine: whether there exists an assignment $\sigma : \{x_i\} \to \{\mathrm{T}, \mathrm{F}\}$ such that $\phi(\sigma) = \mathrm{T}$.

Cook-Levin Reduction: Encode the space-time-state-symbol trajectory of a Turing machine using Boolean variables, constructing the following constraints:

Constraint	Description
Uniqueness	Exactly one state per time step, exactly one symbol per cell
Legal Transition	Adjacent time steps satisfy the transition function
Initial Configuration	Input matches the starting state
Acceptance Condition	There exists a time step in the accepting state

Based on this construction, it can be proven that ϕ_x is satisfiable $\Leftrightarrow x$.

3-SAT Problem and Traditional Solution Difficulties

3-SAT is a restricted version of SAT: each clause contains exactly three literals, and it remains NP-complete *(1972, Karp)*. Traditional silicon-based computer algorithms for solving 3-SAT include backtracking, greedy algorithms, local search, and heuristic random walk algorithms. Although modern SAT solvers perform well on many instances, they still degenerate to exponential time in worst-case scenarios. The known lower bound is $\Omega(1.307^n)$ (2023), while the upper bound is $O^*(1.439^n)$. For random 3-SAT instances with $n \geq 1000$, the solution time remains uncontrollable. This motivates the search for computational paradigms beyond the von Neumann architecture, with DNA computing being one of the most representative non-traditional computational models.

3.2 DNA Computing Model and Its Application Capabilities in SAT Problems

DNA computing modeling involves mapping certain operations or elements of computational problems to DNA molecules and base fragments. Its core idea is to utilize the DNA base pairing principle (A-T, G-C) to encode information and implement computational processes through biochemical operations. The basic DNA computing model includes the following two steps:

- Encoding Strategy: Use DNA sequences to encode problem variables, constraints, and candidate solution information, such as each variable corresponding to a base pair, clauses corresponding to complementary sequences, and candidate solutions corresponding to

sequence fragments or DNA single strands.
- Biochemical Operations: Utilize a series of biochemical operations to simulate logical operations and screening in the computational process, including:
  - Hybridization - achieving logical matching through complementary fragment binding;
  - Enzymatic Cleavage - using restriction endonucleases/exonucleases to remove sequences that do not satisfy conditions;
  - PCR Amplification - replicating specific DNA fragments to increase concentration;
  - Electrophoretic Separation - separating different DNA molecules based on length or mass.

Taking a simple SAT problem as an example: Suppose there are Boolean variables x_1, x_2, x_3, and we need to find an assignment that makes $(x_1 \vee \bar{x}_2) \wedge (\bar{x}_1 \vee x_3)$ true. (1) Encoding Phase: Create DNA sequences representing true/false for each variable, e.g., for variable x_1, use sequence ATCG to represent $x_1 = true$, and GCAT to represent $x_1 = false$. (2) Parallel Generation: By mixing all possible combinations in a solution environment, simultaneously generate all $2^3 = 8$ possible assignment combinations, enabling all candidate solutions to be verified simultaneously. (3) Screening Phase: Design corresponding probes for each clause, confirm whether each clause is satisfied through hybridization with complementary fragments, and then decompose fragments that do not satisfy the requirements of the current time step through enzymatic cleavage operations, gradually screening out solutions that satisfy all constraints.

DNA computing demonstrates unique advantages in solving SAT problems: (1) Massive Parallelism. The greatest advantage of DNA computing lies in its natural parallelism. In a single test tube environment, trillions of DNA molecular reactions can be processed simultaneously, equivalent to parallel exploration of exponentially many candidate solutions. This parallelism theoretically allows computations that originally required exponential time to be completed in constant time. (2) Ultra-High Storage Density. DNA molecules have high information storage density, with 1 gram of DNA capable of storing approximately 215 PB of data. This is very helpful for storing large-scale candidate solutions. (3) Precision of Biochemical Operations. Modern biotechnology can achieve specific DNA operations, including precise hybridization, cleavage, and replication, ensuring the accuracy of the computational process.

Although DNA computing also faces some application challenges, such as slow electrophoresis speeds, complex pre-preparation of DNA strands, and the need for solution biochemical environments, the core concepts of DNA computing, including information encoding, hybridization matching, and parallelism, have inspired many subsequent research works and algorithm optimizations.

4 Classical Examples of DNA Computing Solving SAT Problems

4.1 Qinghua Liu et al. (2000): Surface DNA Computing for Solving 4-Variable 3-SAT Problem

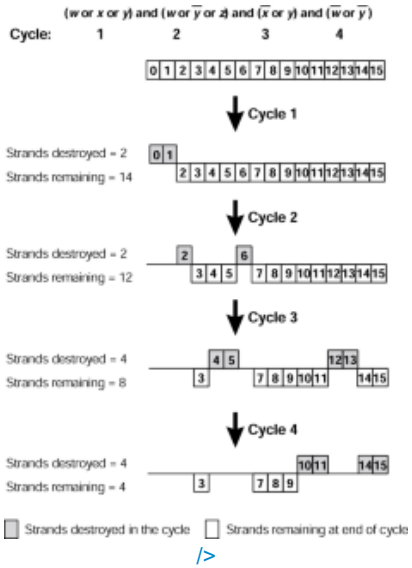
Experimental Overview

This experiment was the first to implement SAT computation on a solid surface (gold film), successfully solving a 3-SAT instance with 4 variables and 4 clauses:

$$\phi = (\bar{x_1} \vee \bar{x_2} \vee x_3) \wedge (x_1 \vee \bar{x_2} \vee \bar{x_3}) \wedge (x_1 \vee x_2 \vee \bar{x_3}) \wedge (\bar{x_1} \vee x_2 \vee x_4)$$

Experimental Principles

1. **Information Encoding**. Synthesize 2^4=16 different single-stranded DNA molecules, with each strand's sequence encoding a possible variable assignment combination; synthesize 4 complementary probe strands, with each probe's sequence storing information about one clause. If a DNA strand satisfies a particular clause, then that DNA strand's sequence is complementary to the clause's sequence.
2. **Cyclic Screening**. As shown in Figure 4.1.1: For each clause in the problem, repeat a set of cycles, simulating cyclic checking of all candidate solutions with individual clauses as step units. (1) **Labeling**: Add probes that are complementary to the DNA sequences satisfying the currently detected clause, thereby hybridizing with DNA strands on the surface encoding "assignment satisfying this clause" information to form double strands, while single strands that fail to hybridize represent those not satisfying the clause. (2) **Destruction**: Add exonuclease to degrade single-stranded DNA, destroying DNA strands not protected by labeling (not satisfying the current clause) in each round. (3) **Delabeling**: Heat to denature double-stranded DNA, remove probes, and restore the remaining DNA on the surface to single-stranded state for the next clause screening.
3. **Result Reading**. After completing all clause cycles, only DNA strands encoding correct answers that satisfy all clauses remain on the surface. Use fluorescent probes to detect the remaining sequences, and decode the answer information encoded in the detection results to obtain the solution.

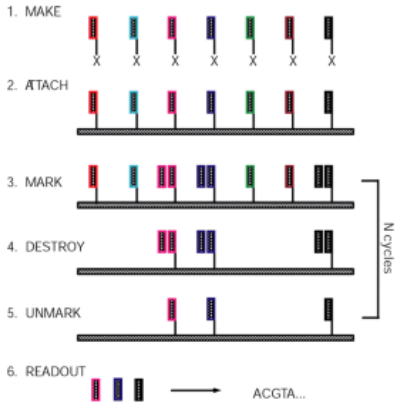


/>

Figure 4.1.1: Schematic diagram of the core cyclic process in Liu et al.'s (2000) experiment

Experimental Procedure

The experimental procedure consists of 6 steps. As shown in Figure 4.1.2: (1) **Preparation**: Represent each possible assignment with a 15-mer single-stranded nucleotide (i.e., 16 possible information strands), and prepare 4 types of probe strands corresponding to the 4 clauses' information; (2) **Immobilization**: Fix all 16 information strands on the gold surface to form a DNA array; (3) **Hybridization**: Sequentially add probes complementary to DNA sequences satisfying the current clause; these probes can "protect" single strands matching the current clause's assignment; (4) **Decomposition**: Add exonuclease Fok I, specifically for degrading single-stranded DNA, destroying DNA single strands not protected by labeling (i.e., not satisfying the current round's clause) in each round; (5) **Delabeling**: Use high temperature to detach probe strands adsorbed onto information strands in step (3); (6) **Reading**: After several cycles of steps (3) to (5), only DNA strands encoding correct answers satisfying all clauses remain on the surface. Use fluorescent-labeled reading probes to detect the remaining sequences, amplify the remaining DNA through PCA, and then determine their sequences to decode the answer.



/>

Figure 4.1.2: Schematic diagram of the surface DNA computing experimental procedure by Liu et al. (2000)

Key technical details: (1) Using thiol-modified DNA sequences to enable firm attachment to the gold film surface; (2) Encoding the T/F state of each variable using specific 15-mer sequences, for example, "x_i is true" is represented by the sequence ATGCGATCGATCGAT; (3) The probe design adopted a "protective" strategy, where only sequences satisfying specific clause conditions can form stable double-stranded structures with the probes, thus being protected in subsequent enzymatic cleavage steps.

Experimental Results

Successfully retained only the assignments satisfying all clauses on the surface, with clear fluorescence signals. After PCA amplification, the concentration of DNA corresponding to the correct answers was significantly higher, successfully identifying four correct answers that satisfied all conditions, as shown in Figure 4.1.3. Experimental data showed that the signal-to-noise ratio between correct and incorrect answers ranged from 10 to 777, allowing clear differentiation.

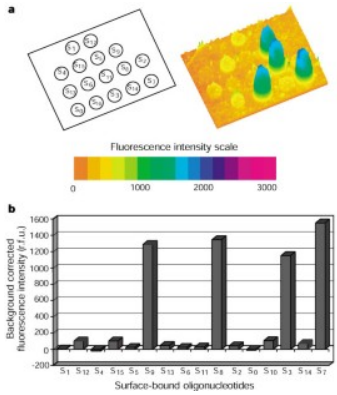


Figure 4.1.3: DNA concentration distribution in surface DNA computing experimental results

Experimental Significance and Criticisms

- Pioneering significance: First demonstration that SAT problems can be solved on solid surfaces in a fully automated, parallel manner; proposed a new paradigm, first demonstration of solving computational problems through programmable biochemical operations on solid surfaces, establishing the "surface DNA computing" branch; demonstrated the high parallelism of DNA computing, processing massive molecules in a single operation (verifying all candidate solutions), showcasing powerful parallel computing capabilities.
- Technical limitations: Each variable in the experiment required a 15-base DNA sequence for encoding, with synthesis difficulty and time consumption increasing dramatically as problem scale expands; subsequent research pointed out potential operational errors, where "marking" steps may be incomplete and "destruction" steps may not be thorough, leading to residual errors. Therefore, critics argue that as problem scale (number of variables) increases, operational errors accumulate, potentially causing satisfiable problems to be misjudged as unsatisfiable.
- Challenges in method scaling: Although this method successfully solved the 4-variable SAT problem, scaling this approach to larger problems faces significant challenges. When the number of variables exceeds 10, experimental complexity grows exponentially, and the cumulative effect of operational errors severely impacts result reliability.

4.2 Ravinderjit S. Braich et al. (2002): Sticker Model for Solving 20-Variable 3-SAT Problem

Experimental Overview

This experiment employed innovative methods including the "Sticker Model" and gel electrophoresis to successfully solve a 3-SAT problem containing 20 variables. (Problem instance and solution are provided at the end of the document)

Experimental Principles

The core principle of this experiment is based on the "Sticker Model," which aims to dynamically encode and manipulate information through reversible nucleic acid hybridization processes. Long single-stranded DNA "memory strands" are constructed, with different short DNA "sticker strands" capable of binding to specific positions on the memory strands. The binding or non-binding of these stickers encodes corresponding Boolean information. Sticker strands bind to memory strands through base complementarity pairing, a process that can be made reversible for "sticking" and "erasing" by altering temperature or chemical environment. The initialized DNA molecular pool, containing all possible solutions, is introduced into a network composed of multiple "separator" modules, as shown in Figure 4.2.1. Each separator module contains a specific sticker DNA designed to capture only DNA molecules that satisfy a particular clause, simulating clause logic gates. Through continuous "filtering" operations, parallel verification of all constraint conditions is completed. The remaining DNA molecules that satisfy all constraints contain the solution information.

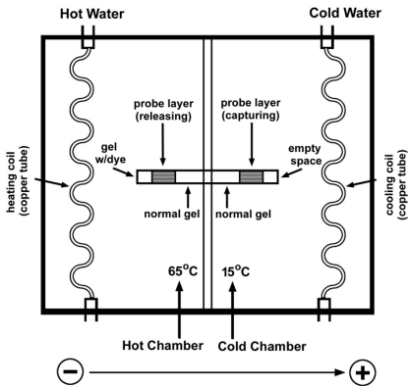


Figure 4.2.1: Separator in the Sticker Model by Braich et al. (2002)

Experimental Procedure

1. **Library Initialization**: Synthesize a library of designed memory strands with fixed length (300 nt). Subsequently, based on the specific form of the SAT problem to be solved, corresponding sticker strands are attached to the memory strands through a series of hybridization steps, thereby constructing an initial candidate solution space representing all 2^{20} possible assignment combinations.
2. **Separator Module Design**: Prepare a series of glass modules filled with polyacrylamide gel. Each module's gel contains immobilized specific DNA probe sequences that encode the satisfaction conditions for one clause in the 3-SAT problem to be solved, designed to be complementary to the memory strand regions corresponding to clauses being satisfied.
3. **Cyclic Screening Computation**: As shown in Figure 4.2.1, the initial library is loaded into the first separator module, initiating the following cyclic process: (1) **Electrophoretic Drive and Specific Capture**: Under electric field, DNA molecules flow directionally between modules. When flowing through a module, DNA strands whose sequences are complementary to the fixed probes in the gel are captured and retained in the gel; unsatisfied strands pass through smoothly into the waste reservoir. (2) **Elution and Transfer**: After completing screening in one module, captured DNA strands are detached from the probes by heating. The remaining filtered liquid, representing the set of solutions satisfying the previous round's clause, is then electrophoretically driven to the next module for the next round of screening.
4. **Output Detection and Analysis**: After screening through all modules, the final outflowing molecules contain only correct solutions satisfying all clauses. The minimal final product is amplified through PCR reaction, and then the corresponding variables are decoded to obtain solutions satisfying all clauses.

Experimental Results

After screening, the variable assignments encoded by the final molecular products completely satisfy all constraint conditions of the given 20-variable 3-SAT problem, as shown in Figure 4.2.2. The experiment successfully identified the unique solution from 1,048,576 possibilities. Although molecular loss occurred due to incomplete capture or elution during the process, the signal-to-noise ratio between effective signals and background noise was sufficient to demonstrate the experiment's validity through careful encoding design and process optimization.

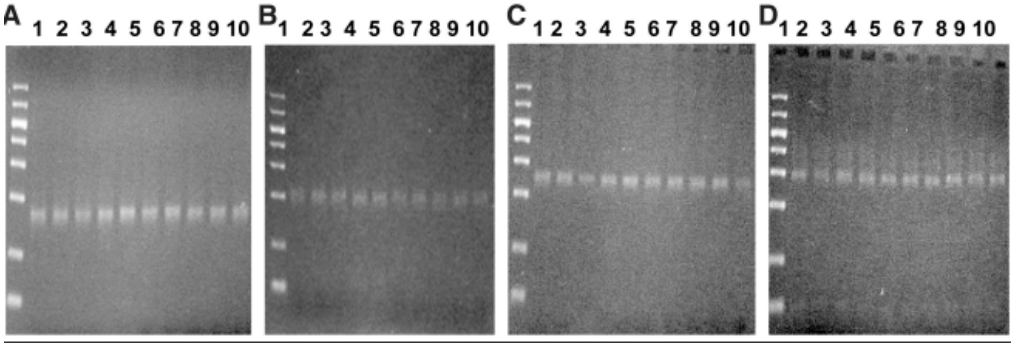


Figure 4.2.2: Signal corresponding to final product encoding in the Sticker Model by Braich et al. (2002)

Experimental Significance and Criticisms

- **Significance**
  - This experiment was the first to solve an NP-complete problem with 20 variables using non-silicon-based methods, significantly advancing the scale of problems that DNA computing can handle, and demonstrating the feasibility of executing complex multi-step discrete algorithms using molecular systems.
  - The "Sticker Model" architecture proposed in this experiment provides a modular, programmable, and automated paradigm for DNA computing, expanding its universality.
  - This work greatly inspired and led DNA computing research for over a decade, stimulating the design of numerous algorithms for solving other NP problems (such as graph coloring, set covering, etc.).
- **Criticisms**
  - Despite solving the 20-variable problem, further scaling faces exponentially increasing challenges. Inherent biochemical errors at each step accumulate progressively, eventually leading to excessively low signal-to-noise ratios and unreliable results. Subsequent analyses indicate that such methods face fundamental limitations when dealing with larger-scale NP-complete problems.
  - The entire computational process takes hours to days, with computation time primarily consumed by slow electrophoretic migration and temperature control cycles, making time efficiency far inferior to electronic computers for solving similar problems.
  - Designing, synthesizing, and validating memory strands and probe sequences for specific problems is itself a tedious task. Complex encoding schemes also increase design difficulty and complexity.

5 Simulation of DNA Computing Using Open-Source Code with Visualization Added

5.1 Python Simulation Framework

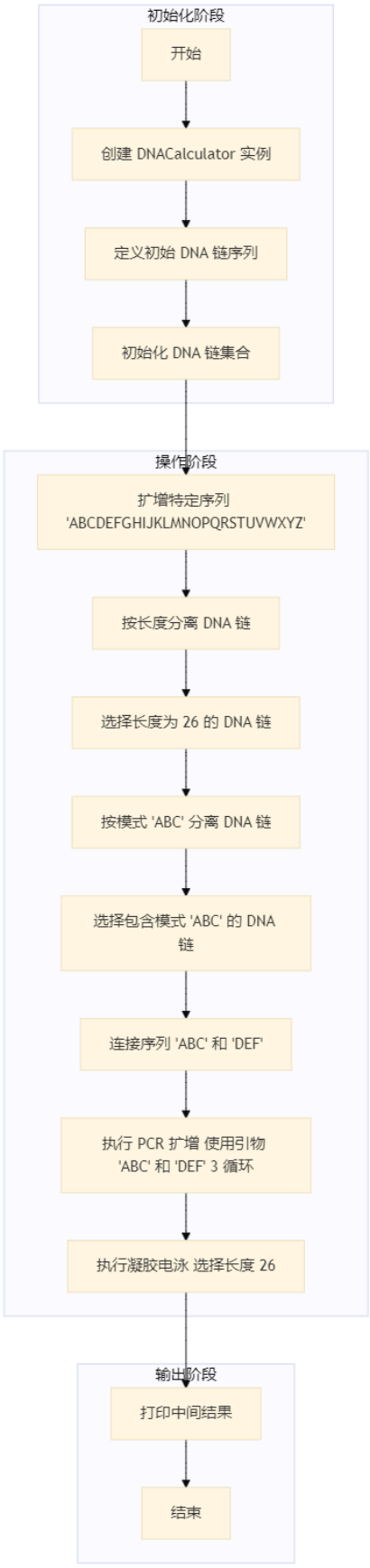


Figure 5.1: the process of Solving SAT problems by simulating DNA computing with original code

[3]This Python simulation framework is designed to simulate key biochemical operations in DNA computing. It employs an object-oriented design and includes operations such as DNA strand initialization, amplification, separation, and screening. The simulation process is as follows:

1. **Initialization**: Create an initial set of DNA strands representing all possible variable assignment combinations (candidate solutions)
2. **Biochemical Operations**: Sequentially execute PCR amplification, separation, screening, and other operations
3. **Result Verification**: Validate the final results through gel electrophoresis and sequence analysis

Main functionalities:

- **Base Operations**: Calculation of complementary sequences
- **Amplification**: Simulation of PCR amplification to increase specific sequences
- **Separation**: Separation of DNA strands based on characteristics such as length
- **Screening**: Filtering of DNA strands based on specific conditions
- **Ligation**: Simulation of DNA ligase function
- **Electrophoresis**: Simulation of gel electrophoresis separation process

Code Execution Flow in Main Function:

```
# Create DNA calculator simulator instance
calc = DNACalculator()
# Define initial DNA strand set, representing different computational inputs
initial_strands = pass
# Initialize DNA strand set
calc.initialize_strands(initial_strands)
# Amplify specific sequences, simulating increased weight of specific inputs
calc.amplify("ABCDEFGHIJKLMNOPQRSTUVWXYZ")
# Separate DNA strands by length
length_groups = calc.separate_by_length()
# Select DNA strands of specific length
selected_by_length = calc.select_by_length(26)
# Separate DNA strands by pattern
with_pattern, without_pattern = calc.separate_by_pattern("ABC")
# Select DNA strands containing specific patterns
selected_by_pattern = calc.select_by_pattern("ABC")
# Ligate two DNA strands, simulating logical operations
ligated = calc.ligate("ABC", "DEF")
# Perform PCR amplification, simulating signal amplification
calc.pcr("ABC", "DEF", cycles=3)
# Perform gel electrophoresis, selecting DNA strands of specific length
calc.gel_electrophoresis(26)
# Print all intermediate results of the entire computation process
calc.print_intermediate_results()
# Visualize DNA strand evolution process
calc.visualize_strand_evolution()
```

5.2 Basic Class Design

The core class is `DNACalculator`, which encapsulates operations such as DNA strand initialization, amplification, separation, and selection.

```
class DNACalculator:
    def __init__(self):
        # Store all current DNA sequences
        self.strands = []
        # Store intermediate results
        self.intermediate_results = []
```

5.3 Low-Level Biochemical Operations

Complementary Sequences and Reverse Complementary Sequences

The simulator implements methods `complement()` and `reverse_complement()` based on the principle of base complementary pairing:

```
# Define DNA base mapping relationships
base_map = {'A': 'T', 'T': 'A', 'C': 'G', 'G': 'C'}
def complement(sequence):
    """Get complementary sequence"""
    return ''.join([base_map.get(base, base) for base in sequence])
def reverse_complement(sequence):
    """Get reverse complementary sequence"""
    return complement(sequence)[::-1]
```

PCR Amplification Simulation

PCR is a key amplification technique in DNA computing, implemented through the `amplify()` method:

```
def amplify(self, target_sequence, fold=2):
    """
    PCR amplification of specific sequences
    Parameters:
        target_sequence (str): Target sequence to be amplified
        fold (int): Amplification factor, default is 2
    """
    count = self.strands.count(target_sequence)
    self.strands.extend([target_sequence] * count * fold)
    self.intermediate_results.append((f'amplify {target_sequence} {fold}x', self.strands.copy()))
```

Separation and Screening Operations

Provides methods for separating DNA strands based on length or pattern:

```
def separate_by_length(self):
    """Separate DNA strands by length"""
    length_groups = {}
    for strand in self.strands:
        length = len(strand)
        if length not in length_groups:
            length_groups[length] = []
        length_groups[length].append(strand)
    return length_groups
```

```
    def separate_by_pattern(self, pattern):
        """Separate DNA strands by pattern"""
        with_pattern = []
        without_pattern = []
        for strand in self.strands:
            if pattern in strand:
                with_pattern.append(strand)
            else:
                without_pattern.append(strand)
        return with_pattern, without_pattern
```

5.4 Advanced Biochemical Operations

DNA Ligation

DNA ligation is used to combine different DNA fragments to create new gene sequences:

```
    def ligate(self, sequence1, sequence2):
        """Ligate two DNA fragments"""
        ligated = sequence1 + sequence2
        self.strands.append(ligated)
        self.intermediate_results.append((f'ligate {sequence1} + {sequence2}', ligated))
        return ligated
```

Gel Electrophoresis

Separate DNA molecules by length through gel electrophoresis:

```
    def gel_electrophoresis(self, length):
        """Simulate gel electrophoresis, screening DNA fragments of specific length"""
        self.strands = [strand for strand in self.strands if len(strand) == length]
        self.intermediate_results.append((f'gel electrophoresis select length {length}', self.strands.copy()))
        return self.strands
```

Complete PCR Process

Complete PCR process including primer binding and fragment amplification:

```
    def pcr(self, primer1, primer2, cycles=10):
        """Simulate complete PCR process"""
        reverse_primer1 = reverse_complement(primer1)
        reverse_primer2 = reverse_complement(primer2)

        for cycle in range(cycles):
            new_strands = []
            for strand in self.strands:
                if primer1 in strand and reverse_primer2 in strand:
                    start_index = strand.find(primer1)
                    end_index = strand.find(reverse_primer2) + len(reverse_primer2)
                    if start_index < end_index:
                        amplified_segment = strand[start_index:end_index]
                        new_strands.append(amplified_segment)
            self.strands.extend(new_strands)
```

5.5 Process Result Tracking

Complete tracking methods for process analysis:

```
    def print_strands(self):
        """Print all current DNA strands and their information"""
        print("Current strands:")
        strand_counts = Counter(self.strands)
        for strand, count in strand_counts.items():
            print(f"  {strand}: {count}")
        print()
```
```python
    def print_intermediate_results(self):
        """Print all intermediate results"""
        print("Intermediate results:")
        for step, result in self.intermediate_results:
            print(f"  Step: {step}")
            pass
```

5.6 Adding Visualization

Two visualizations were added to the original code: total DNA count and the quantity changes of different types of DNA strands, used to clearly display the evolution process of DNA strands at different steps.

```python
def visualize_strand_evolution(self):
        """
        Visualize the evolution process of DNA strands
        """
        if not self.intermediate_results:
            print("No intermediate results to visualize")
            return

        steps = []
        step_names = []
        strand_counts = []
        strand_distributions = []  # Store the count of different DNA sequences at each step

        for i, (step_name, result) in enumerate(self.intermediate_results):
            steps.append(i)
            step_names.append(step_name)

            if isinstance(result, list):
                strand_counts.append(len(result))
                # Count the total number of each DNA sequence type
                counter = Counter(result)
                strand_distributions.append(counter)
            elif isinstance(result, dict):
                # If it's in dictionary format, calculate the total count of all strands
                total_count = 0
                all_strands = []
                for key, value in result.items():
                    if isinstance(value, list):
                        total_count += len(value)
                        all_strands.extend(value)
                strand_counts.append(total_count)
                # Count the total number of each DNA sequence type
                counter = Counter(all_strands)
                strand_distributions.append(counter)
            else:
                strand_counts.append(0)
                strand_distributions.append(Counter())
```

```
# Create charts
fig, (ax1, ax2) = plt.subplots(2, 1, figsize=(14, 12))

# Chart 1: Total DNA strand count changes (line chart)
x = np.arange(len(steps))

ax1.plot(x, strand_counts, marker='o', linestyle='-', linewidth=2,
        label='Total DNA Strands', color='skyblue', markersize=6)

ax1.set_xlabel('Processing Steps')
ax1.set_ylabel('Number of DNA Strands')
ax1.set_title('Total DNA Strands Evolution')
ax1.set_xticks(x)
ax1.set_xticklabels(step_names, rotation=45, ha='right')
ax1.legend()
ax1.grid(True, alpha=0.3)

# Display specific values on the line chart
for i, v in enumerate(strand_counts):
    ax1.text(i, v + 0.5, str(v), ha='center', va='bottom', fontsize=9)



# Chart 2: Count of each DNA strand type (stacked bar chart)
# Get all DNA sequences that have appeared in all steps
all_sequences = set()
for counter in strand_distributions:
    all_sequences.update(counter.keys())
all_sequences = sorted(list(all_sequences))

# Assign colors to each DNA sequence type
colors = plt.cm.Set3(np.linspace(0, 1, len(all_sequences)))

# Prepare stacked data
bottom = np.zeros(len(steps))
for seq_idx, sequence in enumerate(all_sequences):
    counts = []
    for counter in strand_distributions:
        counts.append(counter.get(sequence, 0))

    ax2.bar(step_names, counts, bottom=bottom, label=sequence,
           color=colors[seq_idx % len(colors)], alpha=0.8)
    bottom += np.array(counts)

ax2.set_xlabel('Processing Steps')
ax2.set_ylabel('Number of DNA Strands')
ax2.set_title('DNA Strand Distribution by Type')
ax2.legend(bbox_to_anchor=(1.05, 1), loc='upper left')
ax2.set_xticklabels(step_names, rotation=45, ha='right')
ax2.grid(True, alpha=0.3)

plt.tight_layout()
plt.show()
plt.close()
```

5.7 Simulation Example and Result Display

A complete DNA computing simulation example demonstrating the entire process from initialization to final results with corresponding visualizations:

```
def main():
    calc = DNACalculator()

    # Initialize DNA strand set
    initial_strands = [
        "ABCDEFGHIJKLMNOPQRSTUVWXYZ",      # Input A
        "XYZABCDEFGHIJKLMNOPQRSTUVW",      # Input B
        "ABCDEFGHIJKLMNOPQRSTUVWXY",       # Input C
        "QRSTUVWXYZABCDEFGHIJKLMNOP"       # Input D
    ]

    calc.initialize_strands(initial_strands)
    calc.print_strands()


    calc.amplify("ABCDEFGHIJKLMNOPQRSTUVWXYZ")
    calc.separate_by_length()
    calc.select_by_length(26)
    calc.separate_by_pattern("ABC")
    calc.ligate("ABC", "DEF")
    calc.pcr("ABC", "DEF", cycles=3)
    calc.gel_electrophoresis(26)

    calc.print_intermediate_results()
    #Complete output results are detailed at the end of the document
```

Figure 5.7: Original open-source code simulating DNA computing to solve SAT problem process

5.8 Significance of DNA Computing Simulation

Python simulation of DNA computing has the following important significance: (1) Supports complex SAT problem solving, enabling computers to handle multi-variable SAT problems; (2) Visualizes the computation process: Uses matplotlib visualization to clearly display the evolution process of DNA strands and simulate solution environments; (3) Performance optimization: Algorithm efficiency is relatively high for large-scale problems, eliminating the need to prepare complex DNA molecules and primers; (4) Environmental stability: No need to consider biochemical solution environments, directly computing in simulation, making it convenient and efficient.

6 Subsequent Development and Inspirational Significance of DNA Computing

Since 2002, DNA computing has continued to develop in the field of SAT and more broadly in constraint satisfaction problems, with main directions including:(1)Automation and Microfluidics: DNA logic gate circuits → Fully automated 20-variable SAT solvers.(2)Molecular Programming Languages and Compilers: CRN-to-DNA compilers.(3)Integration with Other Emerging Computing Paradigms: Such as DNA+genetic algorithms, graph neural networks (GNN).(4)Scaling Attempts: Using DNA origami to solve a 70-variable random 3-SAT instance *(Yigong Shi, 2021)*

Although DNA computing currently still cannot compete with top solvers in practical solving of large-scale SAT instances, its theoretical significance is profound: it demonstrates that biological molecules can achieve complex logical computations; massive parallelism provides new ideas for breaking exponential time limitations; it has inspired the design of "global search" mechanisms in quantum computing; it provides a paradigm for interdisciplinary research between biology and computer science; it has inspired the author's thinking about using cellular automata and genetic algorithms to solve logical operations and interest in discrete mathematics. (True!!)

7 References and Related Materials

Liu, Q., Wang, L., Frutos, A. G., Condon, A. E., Corn, R. M., & Smith, L. M. (2000). DNA computing on surfaces. *Nature*, 403(6766), 175–179. https://doi.org/10.1038/35003155

Braich, R. S., Chelyapov, N., Johnson, C., Rothemund, P. W., & Adleman, L. M. (2002). Solution of a 20-variable 3-SAT problem on a DNA computer. *Science*, 296(5567), 499–502. https://doi.org/10.1126/science.1069528

Adleman, L. M. (1994). Molecular computation of solutions to combinatorial problems. Science, *266*(5187), 1021–1024. https://doi.org/10.1126/science.7973651

Zobront, J. (n.d.). DNA computing simulator [Computer software]. GitHub. Retrieved from https://github.com/zobront/dna-computing-simulator

Code and related materials: https://github.com/YCstuding/Discrete-Mathematics-Project-2025

Figure 1: Problems and solutions in the 20-SAT problem

A

Φ =(~x_3 *or* ~x_{16} *or* x_{18}) *and* (x_5 *or* x_{12} *or* ~x_9) *and* (~x_{13} *or* ~x_2 *or* x_{20}) *and* (x_{12} *or* ~x_9 *or* ~x_5) *and* (x_{19} *or* ~x_4 *or* x_6) *and* (x_9 *or* x_{12} *or* ~x_5) *and* (~x_1 *or* x_4 *or* ~x_{11}) *and* (x_{13} *or* ~x_2 *or* ~x_{19}) *and* (x_5 *or* x_{17} *or* x_9) *and* (x_{15} *or* x_9 *or* ~x_{17}) *and* (~x_5 *or* ~x_9 *or* ~x_{12}) *and* (x_6 *or* x_{11} *or* x_4) *and* (~x_{15} *or* ~x_{17} *or* x_7) *and* (~x_6 *or* x_{19} *or* x_{13}) *and* (~x_{12} *or* ~x_9 *or* x_5) *and* (x_{12} *or* x_1 *or* x_{14}) *and* (x_{20} *or* x_3 *or* x_2) *and* (x_{10} *or* ~x_7 *or* ~x_8) *and* (~x_5 *or* x_9 *or* ~x_{12}) *and* (x_{18} *or* ~x_{20} *or* x_3) *and* (~x_{10} *or* ~x_{18} *or* ~x_{16}) *and* (x_1 *or* ~x_{11} *or* ~x_{14}) *and* (x_8 *or* ~x_7 *or* ~x_{15}) *and* (~x_8 *or* x_{16} *or* ~x_{10})

B

x_1=F, x_2=T, x_3=F, x_4=F, x_5=F, x_6=F, x_7=T, x_8=T, x_9=F, x_{10}=T, x_{11}=T, x_{12}=T, x_{13}=F, x_{14}=F, x_{15}=T, x_{16}=T, x_{17}=T, x_{18}=F, x_{19}=F, x_{20}=F

Code block : output results after Python code simulation

```
Initializing strands...
Current strands:
  ABCDEFGHIJKLMNOPQRSTUVWXYZ: 1
  XYZABCDEFGHIJKLMNOPQRSTUVW: 1
  ABCDEFGHIJKLMNOPQRSTUVWXY: 1
  QRSTUVWXYZABCDEFGHIJKLMNOP: 1
```

```
Amplifying specific strands...
Current strands:
ABCDEFGHIJKLMNOPQRSTUVWXYZ: 3
XYZABCDEFGHIJKLMNOPQRSTUVW: 1
ABCDEFGHIJKLMNOPQRSTUVWXY: 1
QRSTUVWXYZABCDEFGHIJKLMNOP: 1


Separating strands by length...
{26: ['ABCDEFGHIJKLMNOPQRSTUVWXYZ',
'XYZABCDEFGHIJKLMNOPQRSTUVW',
'QRSTUVWXYZABCDEFGHIJKLMNOP',
'ABCDEFGHIJKLMNOPQRSTUVWXYZ',
'ABCDEFGHIJKLMNOPQRSTUVWXYZ'], 25:
['ABCDEFGHIJKLMNOPQRSTUVWXY']}


Selecting strands of specific length...
['ABCDEFGHIJKLMNOPQRSTUVWXYZ',
'XYZABCDEFGHIJKLMNOPQRSTUVW',
'QRSTUVWXYZABCDEFGHIJKLMNOP',
'ABCDEFGHIJKLMNOPQRSTUVWXYZ',
'ABCDEFGHIJKLMNOPQRSTUVWXYZ']


Separating strands by pattern...
With pattern 'ABC':
['ABCDEFGHIJKLMNOPQRSTUVWXYZ',
'XYZABCDEFGHIJKLMNOPQRSTUVW',
'ABCDEFGHIJKLMNOPQRSTUVWXY',
'QRSTUVWXYZABCDEFGHIJKLMNOP',
'ABCDEFGHIJKLMNOPQRSTUVWXYZ',
'ABCDEFGHIJKLMNOPQRSTUVWXYZ']
Without pattern 'ABC': []


Selecting strands with specific pattern...
['ABCDEFGHIJKLMNOPQRSTUVWXYZ',
'XYZABCDEFGHIJKLMNOPQRSTUVW',
'ABCDEFGHIJKLMNOPQRSTUVWXY',
'QRSTUVWXYZABCDEFGHIJKLMNOP',
'ABCDEFGHIJKLMNOPQRSTUVWXYZ',
'ABCDEFGHIJKLMNOPQRSTUVWXYZ']


Ligating two strands...
Current strands:
  ABCDEFGHIJKLMNOPQRSTUVWXYZ: 3
  XYZABCDEFGHIJKLMNOPQRSTUVW: 1
  ABCDEFGHIJKLMNOPQRSTUVWXY: 1
  QRSTUVWXYZABCDEFGHIJKLMNOP: 1
  ABCDEF: 1


Performing PCR...
Current strands:
ABCDEFGHIJKLMNOPQRSTUVWXYZ: 3
XYZABCDEFGHIJKLMNOPQRSTUVW: 1
ABCDEFGHIJKLMNOPQRSTUVWXY: 1
QRSTUVWXYZABCDEFGHIJKLMNOP: 1
ABCDEF: 1
```

```
Performing gel electrophoresis...
Current strands:
  ABCDEFGHIJKLMNOPQRSTUVWXYZ: 3
  XYZABCDEFGHIJKLMNOPQRSTUVW: 1
  QRSTUVWXYZABCDEFGHIJKLMNOP: 1


Intermediate results:
  Step: initialize
    ['ABCDEFGHIJKLMNOPQRSTUVWXYZ',
    'XYZABCDEFGHIJKLMNOPQRSTUVW',
    'ABCDEFGHIJKLMNOPQRSTUVWXY',
    'QRSTUVWXYZABCDEFGHIJKLMNOP']


Step: amplify ABCDEFGHIJKLMNOPQRSTUVWXYZ 2x
  ['ABCDEFGHIJKLMNOPQRSTUVWXYZ',
  'XYZABCDEFGHIJKLMNOPQRSTUVW',
  'ABCDEFGHIJKLMNOPQRSTUVWXY',
  'QRSTUVWXYZABCDEFGHIJKLMNOP',
  'ABCDEFGHIJKLMNOPQRSTUVWXYZ',
  'ABCDEFGHIJKLMNOPQRSTUVWXYZ']


Step: separate by length
  26: ['ABCDEFGHIJKLMNOPQRSTUVWXYZ',
  'XYZABCDEFGHIJKLMNOPQRSTUVW',
  'QRSTUVWXYZABCDEFGHIJKLMNOP',
  'ABCDEFGHIJKLMNOPQRSTUVWXYZ',
  'ABCDEFGHIJKLMNOPQRSTUVWXYZ']
  25: ['ABCDEFGHIJKLMNOPQRSTUVWXY']


Step: select length 26
  ['ABCDEFGHIJKLMNOPQRSTUVWXYZ',
  'XYZABCDEFGHIJKLMNOPQRSTUVW',
  'QRSTUVWXYZABCDEFGHIJKLMNOP',
  'ABCDEFGHIJKLMNOPQRSTUVWXYZ',
  'ABCDEFGHIJKLMNOPQRSTUVWXYZ']


Step: separate by pattern ABC
  with: ['ABCDEFGHIJKLMNOPQRSTUVWXYZ',
  'XYZABCDEFGHIJKLMNOPQRSTUVW',
  'ABCDEFGHIJKLMNOPQRSTUVWXY',
  'QRSTUVWXYZABCDEFGHIJKLMNOP',
  'ABCDEFGHIJKLMNOPQRSTUVWXYZ',
  'ABCDEFGHIJKLMNOPQRSTUVWXYZ']
  without: []


Step: select pattern ABC
  ['ABCDEFGHIJKLMNOPQRSTUVWXYZ',
  'XYZABCDEFGHIJKLMNOPQRSTUVW',
  'ABCDEFGHIJKLMNOPQRSTUVWXY',
  'QRSTUVWXYZABCDEFGHIJKLMNOP',
  'ABCDEFGHIJKLMNOPQRSTUVWXYZ',
  'ABCDEFGHIJKLMNOPQRSTUVWXYZ']
```

```
Step: ligate ABC + DEF
Step: pcr with primers ABC, DEF for 3 cycles
  ['ABCDEFGHIJKLMNOPQRSTUVWXYZ',
  'XYZABCDEFGHIJKLMNOPQRSTUVW',
  'ABCDEFGHIJKLMNOPQRSTUVWXY',
  'QRSTUVWXYZABCDEFGHIJKLMNOP',
  'ABCDEFGHIJKLMNOPQRSTUVWXYZ',
  'ABCDEFGHIJKLMNOPQRSTUVWXYZ', 'ABCDEF']


Step: gel electrophoresis select length 26
  ['ABCDEFGHIJKLMNOPQRSTUVWXYZ',
  'XYZABCDEFGHIJKLMNOPQRSTUVW',
  'QRSTUVWXYZABCDEFGHIJKLMNOP',
  'ABCDEFGHIJKLMNOPQRSTUVWXYZ',
  'ABCDEFGHIJKLMNOPQRSTUVWXYZ']
```


1. https://doi.org/10.1038/35003155 ↩
2. https://doi.org/10.1126/science.1069528 ↩
3. https://github.com/zobront/dna-computing-simulator ↩