CPSC-406 Report

Jaime Song
Chapman University

02/16/25

Abstract

Contents

1 Introduction

This report brings together a series of problems and exercises that reflect a deeper engagement with the foundational ideas of theoretical computer science. Each section builds on prior concepts, combining formal reasoning, algorithmic thinking, and structured problem solving. The goal is to demonstrate a clear understanding of the material through worked examples, precise explanations, and methodical approaches to each challenge. The report emphasizes clarity, logic, and rigor in both presentation and analysis.

2 Week by Week

2.1 Week 1

Homework 1
Introduction to Automata Theory

Characterize all the accepted words (i.e., describe exactly those words that get recognized).

A word is accepted if it consists of the symbols 5 and 10, and the sum of all symbols is exactly 25.

$$\begin{aligned} \text{Let } a &= \text{number of times 5 appears} \\ b &= \text{number of times 10 appears} \\ a, b &\in \mathbb{N} \\ 5a + 10b &= 25 \end{aligned}$$

Valid (a, b) combinations:

- (5, 0): 5, 5, 5, 5, 5
- (3, 1): 5, 5, 5, 10
- (1, 2): 5, 10, 10

Characterize all the accepted words. Can you describe them via a regular expression?

A word is accepted if it consists of one or more "pay" actions followed by a "push" action. This sequence can repeat any number of times.

$$(\texttt{pay}^{+}\ \texttt{push})^{+}$$

This ensures that each ‘push‘ is preceded by one or more ‘pay‘s, and the entire pattern can repeat.

2.2 Week 2

Homework 1
Deterministic and Non-deterministic Finite Automata (DFAs and NFAs)

Determine for the following words if they are contained in L1, L2, or L3.

Word	L1	L2	L3
10011	Yes	No	No
100	No	No	No
10100100	Yes	Yes	Yes
1010011100	Yes	No	No
11110000	No	Yes	Yes

Consider the paths corresponding to the words w1 = 0010, w2 = 1101, and w3 = 1100. For which of these words does their run end in the accepting state?

- **Accepted:** w1 = 0010, w2 = 1101
- **Not Accepted:** w3 = 1100

ITALC Chapter 2.1 Summary
This section introduces finite automata through a real-world example: electronic money. The challenge is ensuring that digital currency can't be forged, duplicated, or spent multiple times. A bank plays a crucial role by issuing encrypted money files and keeping track of all valid transactions. While cryptography secures the money itself, protocols must prevent fraud and ensure transactions follow the intended process. The system involves three participants: the customer, the store, and the bank. The customer can either pay the store or cancel the transaction, returning the money to their account. The store can ship goods or redeem the money by sending it to the bank, which then transfers ownership. However, issues arise if the customer attempts to both pay and cancel, or if the store ships goods before confirming payment. To analyze these interactions, each participant's behavior is modeled as a finite automaton, where states represent different stages of a transaction and transitions correspond to actions taken. When combining these automata, the overall system can be examined for vulnerabilities. This method reveals flaws in poorly designed protocols, such as scenarios where the store ships goods but never gets paid. By using automata, we can validate protocols and ensure secure digital transactions.

2.3 Week 2

Homework 2
Programming (with) Automata

Code for Exercise 2

```
dfa.py:
# a class for DFAs
class DFA :
    # init the DFA
    def __init__(self, Q, Sigma, delta, q0, F) :
        self.Q = Q # set of states
        self.Sigma = Sigma # set of symbols
        self.delta = delta # transition function
        self.q0 = q0 # initial state
        self.F = F # final states

   # print the data of the DFA
    def __repr__(self) :
        return f"DFA({self.Q},\n\t{self.Sigma},\n\t{self.delta},\n\t{self.q0},\n\t{self.F})"

    # run the DFA on the word w
    # return if the word is accepted or not
    def run(self, w) :
        current_state = self.q0 # start at initial state
        for symbol in w: # process each character in the input word
            if (current_state, symbol) in self.delta:
                current_state = self.delta[(current_state, symbol)] # Move to next state
            else:
                return False # invalid transition, reject the word
        return current_state in self.F # accept if final state is in F
```

dfa_ex01.py:

```
import dfa
# generate words for testing
def generate_words():
    words = []
    alphabet = ['a', 'b']
    for first in alphabet:
        for second in alphabet:
            for third in alphabet:
                words.append(first + second + third)
    return words

def __main__() :
    Q1 = {0,1}
    Sigma1 = {'a', 'b'}
    delta1 = {(0, 'a'): 1, (0, 'b'): 0, (1, 'a'): 1, (1, 'b'): 0}
    q01 = 0
    F1 = {1}
    A1 = dfa.DFA(Q1, Sigma1, delta1, q01, F1)

    Q2 = {0, 1}
    Sigma2 = {'a', 'b'}
    delta2 = {(0, 'a'): 0, (0, 'b'): 1, (1, 'a'): 1, (1, 'b'): 1}
    q02 = 0
    F2 = {1}
    A2 = dfa.DFA(Q2, Sigma2, delta2, q02, F2)

    words = generate_words()
    automata = [A1, A2]

    # test words on automata
    for X in automata:
         print(f"{X.__repr__()}")
         for w in words:
            print(f"{w}: {X.run(w)}")
         print("\n")

__main__()
```

output:

```
DFA({0, 1}, {'b', 'a'}, {(0, 'a'): 1, (0, 'b'): 0, (1, 'a'): 1, (1, 'b'): 0}, 0, {1})
aaa: True
aab: False
aba: True
abb: False
baa: True
bab: False
bba: True
bbb: False


DFA({0, 1}, {'b', 'a'}, {(0, 'a'): 0, (0, 'b'): 1, (1, 'a'): 1, (1, 'b'): 1}, 0, {1})
aaa: False
aab: True
```

```
aba: True
abb: True
baa: True
bab: True
bba: True
bbb: True
```

Exercise 4

```
Accepting State: 4
The DFA accepts any string that contains at least one 'b' since reaching state 4 via any 'b' will
    keep it in an accepting state
```

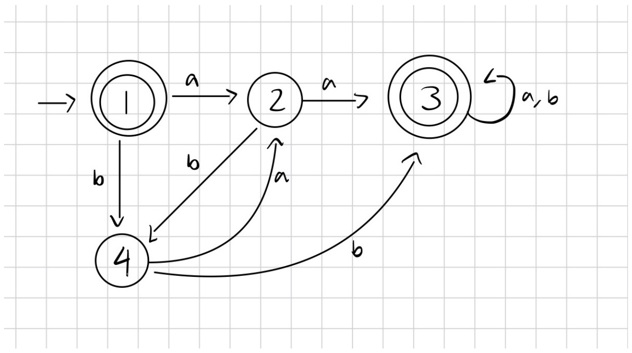

Figure 1:

```
dfa.py:
# a class for DFAs
class DFA :
    # init the DFA
    def __init__(self, Q, Sigma, delta, q0, F) :
        self.Q = Q # set of states
        self.Sigma = Sigma # set of symbols
        self.delta = delta # transition function
        self.q0 = q0 # initial state
        self.F = F # final states

   # print the data of the DFA
    def __repr__(self) :
        return f"DFA({self.Q},\n\t{self.Sigma},\n\t{self.delta},\n\t{self.q0},\n\t{self.F})"

    # run the DFA on the word w
    # return if the word is accepted or not
    def run(self, w) :
        current_state = self.q0 # start at initial state
        for symbol in w: # process each character in the input word
            if (current_state, symbol) in self.delta:
                current_state = self.delta[(current_state, symbol)] # Move to next state
            else:
                return False # invalid transition, reject the word
        return current_state in self.F # accept if final state is in F
    def refuse(self):
        new_final_states = self.Q - self.F # complement final states
        return DFA(self.Q, self.Sigma, self.delta, self.q0, new_final_states)

dfa_ex03.py:
import dfa
def __main__() :
    Q = {1, 2, 3, 4} # set of states
    Sigma = {'a', 'b'} # alphabet
    delta = {
        (1, 'a'): 2, (1, 'b'): 4,
        (2, 'a'): 3, (2, 'b'): 4,
        (3, 'a'): 3, (3, 'b'): 3,
        (4, 'a'): 4, (4, 'b'): 4
    }
    q0 = 1 # initial state
    F = {4} # accepting states

    A = dfa.DFA(Q, Sigma, delta, q0, F) # instantiate DFA A

    A0 = A.refuse()
    test_cases = ["", "a", "aa", "aaa", "b", "ab", "ba", "abb", "aab"]

    print("Testing A (accepts strings with at least one 'b'):")
    for test in test_cases:
        print(f"String '{test}': {'Accepted' if A.run(test) else 'Rejected'}")

    print("\nTesting A0 (accepts strings without 'b'):")
    for test in test_cases:
        print(f"String '{test}': {'Accepted' if A0.run(test) else 'Rejected'}")

__main__()

output:
Testing A (accepts strings with at least one 'b'):
String '': Rejected
String 'a': Rejected
String 'aa': Rejected
String 'aaa': Rejected
String 'b': Accepted
String 'ab': Accepted
String 'ba': Accepted
String 'abb': Accepted
String 'aab': Rejected

Testing A0 (accepts strings without 'b'):
String '': Accepted
String 'a': Accepted
String 'aa': Accepted
String 'aaa': Accepted
String 'b': Rejected
String 'ab': Rejected
String 'ba': Rejected
String 'abb': Rejected
String 'aab': Accepted
```

ITALC: Exercise 2.2.4

```
a. Set of strings ending in 00
States:
* q0: Start state, accepts anything so far
* q1: Last character was 0
* q2: Last two characters were 00 (accepting state)

Transitions:
* From q0:
0 -> q1, 1-> q0
* From q1:
0 -> q2, 1-> q0
* From q2:
0 -> q2, 1-> q0

Accepting state: q2

b. Set of strings with three consecutive 0's
States:
* q0: Start state, no 0's seen yet
* q1: One 0 seen
* q2: Two consecutive 0's seen
* q3: Three consecutive 0's seen

Transitions:
* From q0:
0 -> q1, 1-> q0
* From q1:
0 -> q2, 1-> q0
* From q2:
0 -> q3, 1-> q0
* From q3:
0 -> q3, 1-> q3

Accepting state: q3

c. Set of strings with 011 as a substring
States:
* q0: Start state, no part of 011 seen yet
* q1: 0 seen
* q2: 01 seen
* q3: 011 seen (accepting state)

Transitions:
* From q0:
0 -> q1, 1-> q0
* From q1:
0 -> q1, 1-> q2
* From q2:
0 -> q1, 1-> q3
* From q3:
0 -> q3, 1-> q3

Accepting state: q3
```

Question:

```
How does a DFA remember whether it has seen the substring '01' during string processing, and how
    do the different states (q_0, q_1, q_2) represent the conditions required for acceptance?
```

2.4 Week 3

Homework 3
Operations on Automata

HW 1: Extended Transition Function

1. Describe the language accepted by $A^{(2)}$:

Automaton $A^{(2)}$ accepts all strings that end in the state 2, which is the only accepting state. From the diagram, this means the string must cause the automaton to go from state 1 to 2 using an odd number of transitions between 1 and 2 on input a, and not end in state 3 (which has a self-loop). Thus, the language consists of strings that contain an odd number of a's causing a final transition into state 2 and do not end in b after going to state 3.

More precisely, $L(A^{(2)})$ contains strings where the final character a transitions to state 2, and earlier transitions may involve state 1 or 3, but must finish at state 2.

2. Compute $\hat{\delta}^{(1)}(1, abaa)$ (for $A^{(1)}$):

From the diagram:

$$\begin{aligned}\delta^{(1)}(1, a) &= 2\\ \delta^{(1)}(2, b) &= 4\\ \delta^{(1)}(4, a) &= 2\\ \delta^{(1)}(2, a) &= 3\end{aligned}$$

Therefore:

$$\hat{\delta}^{(1)}(1, abaa) = \delta^{(1)}(2, a) = 3$$

Since state 3 is not accepting, the string is **not accepted**.

3. Compute $\hat{\delta}^{(2)}(1, abba)$ (for $A^{(2)}$):

From the diagram:

$$\delta^{(2)}(1,a) = 2$$
$$\delta^{(2)}(2,b) = 1$$
$$\delta^{(2)}(1,b) = 3$$
$$\delta^{(2)}(3,a) = 3$$

Therefore:

$$\hat{\delta}^{(2)}(1, abba) = \delta^{(2)}(3,a) = 3$$

Since state 3 is not accepting, the string is **not accepted**.

HW 2: Product Automaton

1. Construct the intersection automaton A for $A^{(1)}$ and $A^{(2)}$:

States of A are ordered pairs (q_1, q_2) where $q_1 \in Q^{(1)}$, $q_2 \in Q^{(2)}$. Transitions follow:

$$\delta((p,q),a) = (\delta^{(1)}(p,a), \delta^{(2)}(q,a))$$

You only need to draw reachable states from the start state $(1,1)$. Final states are $F = F^{(1)} \times F^{(2)}$, i.e., both components must be in final states.

2. Why is $L(A) = L(A^{(1)}) \cap L(A^{(2)})$?

By construction, a string is accepted by the product automaton A only if it drives both component automata $A^{(1)}$ and $A^{(2)}$ to their respective accepting states. Thus, the product automaton simulates both machines in parallel, and only accepts if both would accept individually.

$$L(A) = L(A^{(1)}) \cap L(A^{(2)})$$

3. How to modify A to construct A' such that $L(A') = L(A^{(1)}) \cup L(A^{(2)})$?

To accept the union of the languages, we redefine the set of final states to accept if either automaton accepts:

$$F' = (F^{(1)} \times Q^{(2)}) \cup (Q^{(1)} \times F^{(2)})$$

This means A' accepts a string if **either** $A^{(1)}$ or $A^{(2)}$ accepts it.

ITALC Section 2.3 Question:
How does a nondeterministic finite automaton (NFA) process the string "00101" to accept it, and how does it differ from a deterministic finite automaton (DFA) in terms of state transitions?

2.5 Week 4

Homework 4
Determinization of NFAs

HW 1:
1. Explain how you can view $\mathcal{A}$ as an NFA.

Every DFA is a special case of an NFA. In a DFA, the transition function

$$\delta : Q \times \Sigma \to Q$$

maps each state and input to a single state, while in an NFA, the transition function is defined as

$$\delta' : Q \times \Sigma \to \mathcal{P}(Q),$$

where $\mathcal{P}(Q)$ is the power set of Q, i.e., a set of possible next states.

To interpret a DFA $\mathcal{A}$ as an NFA, we define a new transition function δ' such that:

$$\delta'(q, a) = \{\delta(q, a)\}$$

for every $q \in Q$, $a \in \Sigma$. This simply wraps each deterministic transition in a singleton set.

Thus, the same DFA can be viewed as an NFA $\mathcal{A}' = (Q, \Sigma, \delta', q_0, F)$ where:

- The set of states and alphabet remain unchanged
- The start state and accepting states remain the same
- The transition function now returns sets instead of single values, but the accepted language does not change

Therefore, $L(\mathcal{A}) = L(\mathcal{A}')$.

2. General construction: define an NFA $\mathcal{A}'$ from a DFA $\mathcal{A} = (Q, \Sigma, \delta, q_0, F)$ such that $L(\mathcal{A}') = L(\mathcal{A})$.

Define $\mathcal{A}' = (Q', \Sigma, \delta', q_0', F')$, where:

$$\begin{aligned} Q' &= Q \\ q_0' &= q_0 \\ F' &= F \\ \delta'(q, a) &= \{\delta(q, a)\} \quad \text{for all } q \in Q,\ a \in \Sigma \end{aligned}$$

This construction ensures that the NFA simulates exactly the same transitions as the DFA but within the more general NFA framework.

3. Extended transition function computation and example strings

---

Let us compute the extended transition $\hat{\delta}$ for two example strings:

Let $\mathcal{A}$ be the DFA from the diagram. The transitions are:

- $\delta(1, a) = 2$

- $\delta(1, b) = 1$
- $\delta(2, a) = 2$
- $\delta(2, b) = 3$
- $\delta(3, a) = 3$, $\delta(3, b) = 3$

Now compute:

$$\hat{\delta}(1, abab) = \delta(3, b) = 3 \quad \Rightarrow \text{Accepted}$$

If $v = aba$, we get:

$$\hat{\delta}(1, aba) = \delta(3, a) = 3 \quad \Rightarrow \text{Accepted}$$

And for $w = abba$:

$$\hat{\delta}(1, abba) = \delta(3, a) = 3 \quad \Rightarrow \text{Accepted}$$

Because state 3 is accepting, all of these strings are in $L(\mathcal{A})$.

HW 2:
Language Description and Specification of the NFA

The NFA A accepts binary strings that contain the substring "100". The accepting state q_3 is reached after reading this substring, and from there, the automaton accepts any further symbols via self-loops.

Formal specification of $A = (Q, \Sigma, \delta, q_0, F)$:

- $Q = \{q_0, q_1, q_2, q_3\}$
- $\Sigma = \{0, 1\}$
- Transition function δ:

$$\begin{aligned}
\delta(q_0, 0) &= \{q_0\} \\
\delta(q_0, 1) &= \{q_0, q_1\} \\
\delta(q_1, 0) &= \{q_2\} \\
\delta(q_1, 1) &= \{q_1\} \\
\delta(q_2, 0) &= \{q_3\} \\
\delta(q_2, 1) &= \{q_1\} \\
\delta(q_3, 0) &= \{q_3\} \\
\delta(q_3, 1) &= \{q_3\}
\end{aligned}$$

- Start state: q_0
- Final states: $F = \{q_3\}$

Extended Transition Function: Compute $\hat{\delta}(q_0, 10110)$ step by step.

- Step 1: $\delta(q_0, 1) = \{q_0, q_1\}$
- Step 2: $\delta(\{q_0, q_1\}, 0) = \{q_0, q_2\}$
- Step 3: $\delta(\{q_0, q_2\}, 1) = \{q_0, q_1\}$
- Step 4: $\delta(\{q_0, q_1\}, 1) = \{q_0, q_1\}$
- Step 5: $\delta(\{q_0, q_1\}, 0) = \{q_0, q_2\}$

Result: The final state set is $\{q_0, q_2\}$, which does **not** include q_3, so the string is **not accepted**.

Paths for $v = 1100$ and $w = 1010$:

For $v = 1100$:

- $\delta(q_0, 1) = \{q_0, q_1\}$
- From q_0, reading 1 again $\to \{q_0, q_1\}$
- From q_1, reading 0 $\to q_2$
- From q_2, reading 0 $\to q_3$ (accepting)

Accepted (there exists a path to q_3)

For $w = 1010$:

- $\delta(q_0, 1) = \{q_0, q_1\}$
- From q_0, reading 0 $\to q_0$
- From q_1, reading 0 $\to q_2$
- From q_2, reading 1 $\to q_1$
- From q_0, q_2, q_1, reading 0 $\to \{q_0, q_2\}$

Rejected (no path ends at q_3)

Power Set Construction – Determinization A^D:

Construct DFA A^D where:

- States are subsets of Q
- Start state: $\{q_0\}$
- Accepting states: Any set containing q_3
- Transitions: Defined by union of transitions from each state in the subset

Example states in A^D:

- $\{q_0\}$
- $\{q_0, q_1\}$
- $\{q_0, q_2\}$
- $\{q_3\}$
- $\{q_0, q_2, q_3\}$
- etc.

Verify that $L(A) = L(A^D)$:

Each string accepted by the NFA has a corresponding run in the DFA on the subset states. The DFA constructed via the power set construction simulates all nondeterministic behaviors in parallel, and accepts a string if any computation path ends in an accepting state. Thus, $L(A^D) = L(A)$.

Is there a smaller DFA?

Yes, many subset states of A^D are unreachable or equivalent. For example, $\{q_0, q_2, q_3\}$ and $\{q_0, q_3\}$ behave similarly after reaching q_3, because the machine stays in q_3 indefinitely. So states containing q_3 can be collapsed in a minimized version.

ITALC Section 3.1, 3.2.1, 3.2.2 Question:
What does the Kleene star operation represent in the context of regular expressions?

2.6 Week 5

Homework 5
ITALC Section 3.2 – State Elimination and Regular Expressions

---

3.2.1 (a) – Initial Expressions $R_{ij}^{(0)}$

Here, $R_{ij}^{(0)}$ represents the regular expression describing transitions from state q_i to q_j using zero intermediate states.

$$\begin{aligned}
R_{11}^{(0)} &= \emptyset && \text{(No loop on } q_1\text{)}\\
R_{12}^{(0)} &= 0 && \text{(Transition from } q_1 \to q_2 \text{ on 0)}\\
R_{13}^{(0)} &= \emptyset && \text{(No direct } q_1 \to q_3\text{)}\\
R_{21}^{(0)} &= 1 && \text{(Transition from } q_2 \to q_1 \text{ on 1)}\\
R_{22}^{(0)} &= \emptyset && \text{(No loop on } q_2\text{)}\\
R_{23}^{(0)} &= 0 && \text{(Transition from } q_2 \to q_3 \text{ on 0)}\\
R_{31}^{(0)} &= \emptyset && \text{(No } q_3 \to q_1\text{)}\\
R_{32}^{(0)} &= 1 && \text{(Transition from } q_3 \to q_2 \text{ on 1)}\\
R_{33}^{(0)} &= 0 && \text{(Loop on } q_3 \text{ on 0)}
\end{aligned}$$

3.2.1 (b) – Expressions with One Intermediate State

Using only state q_1 as an intermediate:

$$
\begin{aligned}
R_{11}^{(1)} &= \emptyset \cup (0 \cdot 1) = 01\\
R_{12}^{(1)} &= 0 \cup (0 \cdot \emptyset) = 0\\
R_{13}^{(1)} &= \emptyset \cup (0 \cdot 0) = 00\\
R_{21}^{(1)} &= 1 \cup (\emptyset \cdot 1) = 1\\
R_{22}^{(1)} &= \emptyset \cup (\emptyset \cdot \emptyset) = \emptyset\\
R_{23}^{(1)} &= 0 \cup (\emptyset \cdot 0) = 0\\
R_{31}^{(1)} &= \emptyset \cup (1 \cdot 1) = 11\\
R_{32}^{(1)} &= 1 \cup (1 \cdot \emptyset) = 1\\
R_{33}^{(1)} &= 0 \cup (1 \cdot 0) = 10
\end{aligned}
$$

3.2.1 (c) – Expressions with Two Intermediate States

Using q_1 and q_2 as intermediates:

$$
\begin{aligned}
R_{11}^{(2)} &= 01 \cup (0 \cdot 0 \cdot 11) = 01 \cup 0011\\
R_{12}^{(2)} &= 0 \cup (0 \cdot 0 \cdot 1) = 0 \cup 001 = 0\\
R_{13}^{(2)} &= 00 \cup (0 \cdot 0 \cdot 0) = 00 \cup 000 = 00\\
R_{21}^{(2)} &= 1 \cup (0 \cdot 1 \cdot 11) = 1 \cup 011 = 1\\
R_{22}^{(2)} &= \emptyset \cup (0 \cdot 1 \cdot 1) = \emptyset\\
R_{23}^{(2)} &= 0 \cup (0 \cdot 1 \cdot 0) = 0\\
R_{31}^{(2)} &= 11 \cup (1 \cdot 0 \cdot 11) = 11 \cup 1011 = 11\\
R_{32}^{(2)} &= 1 \cup (1 \cdot 0 \cdot 1) = 1 \cup 101 = 1\\
R_{33}^{(2)} &= 10 \cup (1 \cdot 0 \cdot 0) = 10 \cup 100 = 10
\end{aligned}
$$

3.2.1 (d) – Final Regular Expression for the Language

We aim to describe all strings leading to the accepting state q_3. Based on transitions and paths derived above, the strings that reach q_3 must include sequences like 00 or 000. A valid regular expression capturing this behavior is:

$$(0^*1^*0^*0)$$

—

ITALC Section 4.4 – DFA Minimization

4.4.1 – Distinguishability Table

Step 1: Initial table

	q_1	q_2	q_3
q_1	ϵ		
q_2		ϵ	
q_3			ϵ

Step 2: Mark distinguishable pairs Since q_3 is the only accepting state:

- Mark (q_1, q_3) and (q_2, q_3)

Step 3: Final refined table

	q_1	q_2	q_3
q_1	ϵ	Indist.	Dist.
q_2	Indist.	ϵ	Dist.
q_3	Dist.	Dist.	ϵ

4.4.1 (b) – Reduced DFA Construction

- Merge q_1 and q_2 (indistinguishable) - Keep q_3 separate

New states: - $[q_1, q_2]$ - q_3

Transitions:

- $[q_1, q_2] \xrightarrow{0} [q_1, q_2]$
- $[q_1, q_2] \xrightarrow{1} q_3$
- $q_3 \xrightarrow{0} [q_1, q_2]$
- $q_3 \xrightarrow{1} [q_1, q_2]$

Start state: $[q_1, q_2]$, **Accepting state:** q_3

—

ITALC Section 4.4.2 – Larger DFA Minimization

Distinguishability Table (Partial)

	A	B	C	D
A	ϵ	Indist.	Dist.	Dist.
B	Indist.	ϵ	Dist.	Indist.
C	Dist.	Dist.	ϵ	Dist.
D	Dist.	Indist.	Dist.	ϵ

Equivalence Classes and Reduced DFA

- $[C, F, I]$: all accepting, indistinguishable
- $[D, G]$: equivalent
- $[E, H]$: equivalent

New states: - $[A]$, $[B]$, $[C, F, I]$, $[D, G]$, $[E, H]$

Transitions:

- $[A] \xrightarrow{0} [B]$, $[A] \xrightarrow{1} [E, H]$
- $[B] \xrightarrow{0} [C, F, I]$, $[B] \xrightarrow{1} [E, H]$
- $[C, F, I] \xrightarrow{0} [D, G]$, $[C, F, I] \xrightarrow{1} [E, H]$
- $[D, G] \xrightarrow{0} [E, H]$, $[D, G] \xrightarrow{1} [C, F, I]$

- $[E, H] \xrightarrow{0} [A], \quad [E, H] \xrightarrow{1} [C, F, I]$

Start state: $[A]$, **Accepting state:** $[C, F, I]$

—

ITALC 3.2 and 4.4 Question:

How does the minimization of a DFA impact its computational efficiency, and are there cases where minimizing a DFA could be detrimental rather than beneficial?

2.7 Week 8

Homework 6 and 7
Computability, Turing machines, (un)decidability
Exercise A

1. Write a Turing Machine to accept the language of binary strings $L = \{10^n : n \in \mathbb{N}\}$, and for input 10^n, it returns the string 10^{n+1}.
   - Start state: q_0
   - Accepting state: none (the machine halts naturally after writing)
   - Alphabet: $\Sigma = \{0, 1\}$, Tape symbols include B

Current State	Read	Write	Move	Next State
q_0	1	1	R	q_0
q_0	0	0	R	q_0
q_0	B	0	N	halt

2. Write a Turing Machine to accept the language of binary strings $L = \{10^n : n \in \mathbb{N}\}$, and for input 10^n, it returns the string 1.
   - Start state: q_0
   - Accepting state: none
   - Alphabet: $\Sigma = \{0, 1\}$, Tape symbols include B

Current State	Read	Write	Move	Next State
q_0	1	1	R	q_1
q_1	0	B	R	q_1
q_1	B	B	N	halt

3. Write a Turing Machine to accept the total language of binary strings and, for every string, return the string with 0's and 1's swapped.
   - Start state: q_0
   - Accepting state: none
   - Alphabet: $\Sigma = \{0, 1\}$, Tape symbols include B

Current State	Read	Write	Move	Next State
q_0	0	1	R	q_0
q_0	1	0	R	q_0
q_0	B	B	N	halt

Problems on Decidability

Exercise 1

Which of the following languages are decidable, recursively enumerable (r.e.), or have recursively enumerable complement (co-r.e.)?

---

1. $L_1 := \{M \mid M \text{ halts on itself}\}$

   Answer: This is the self-halting problem. It is not decidable because it reduces to the Halting Problem, which is undecidable. However, it is recursively enumerable (r.e.) since we can simulate M on input M, and accept if it halts.

   Conclusion: Not decidable, r.e., not co-r.e.

2. $L_2 := \{(M, w) \mid M \text{ halts on the word } w\}$

   Answer: This is the classic Halting Problem. It is not decidable, but it is r.e. because we can simulate M on w and accept if it halts.

   Conclusion: Not decidable, r.e., not co-r.e.

3. $L_3 := \{(M, w, k) \mid M \text{ halts on } w \text{ in at most } k \text{ steps}\}$

   Answer: This language is decidable. We can simulate M on w for at most k steps and check if it halts within that time.

   Conclusion: Decidable, hence also r.e. and co-r.e.

Exercise 2

Which of the following statements is true? If yes, give an argument; if no, give a counterexample.

---

1. If L_1 and L_2 are decidable, then so is $L_1 \cup L_2$.

   Answer: True. We can construct a decider that runs both deciders for L_1 and L_2. If either accepts, accept.

2. If L is decidable, then so is the complement $\overline{L} := \Sigma^* \setminus L = \{w \in \Sigma^* \mid w \notin L\}$.

   Answer: True. Since a decider always halts, we can simply invert the result (accept becomes reject and vice versa).

3. If L is decidable, then so is L^*.

   Answer: True. Given a decider for L, we can construct a decider for L^* by checking all possible segmentations of the input string into words from L, which is guaranteed to halt since L is decidable.

4. If L_1 and L_2 are r.e., then $L_1 \cup L_2$ is r.e.

   Answer: True. Run both Turing machines for L_1 and L_2 in parallel (dovetailing). If either accepts, accept.

5. If L is r.e., then so is $\overline{L}$.

   Answer: False. If both a language and its complement are r.e., then the language is decidable. There exist r.e. languages whose complement is not r.e. (e.g., the Halting Problem).

6. If L is r.e., then so is L^*.

   Answer: True. Given an r.e. machine for L, we can build a machine for L^* using non-determinism to guess how to split the input and simulate each piece. Since TMs are allowed to be non-deterministic when recognizing r.e. languages, this works.

ITALC 8.1, 8.2, 9.1, 9.2 Questions:

- Why are Turing machines used instead of actual programming languages when discussing undecidability?
- What makes the diagonalization language L_d so important in proving that some languages are not even recursively enumerable?

2.8 Week 9 and Week 10

Homework 8 and 9
Landau Notation ("Big O"), or Mathematics as a Metaphor
Exercise 1

Let log denote the natural logarithm, i.e., the logarithm w.r.t. the base e. Order the following functions by their order of growth (from slow to fast):

1. (f) $\log(\log n)$
2. (c) $\log n$
3. (b) n
4. (e) n^2
5. (g) 2^n
6. (d) e^n
7. (h) $n!$
8. (a) 2^{2^n}

Exercise 2

For functions $f, g, h \colon \mathbb{N} \to \mathbb{R}_{\geq 0}$, prove the following:

1. $f \in O(f)$

   Proof: By definition, $f(n) \leq c \cdot f(n)$ for any $c \geq 1$ and all $n \geq 1$. Choosing $c = 1$, the inequality holds trivially. Hence, $f \in O(f)$.

2. $O(c \cdot f) = O(f)$ for $c > 0$

   Proof: Multiplying a function by a constant does not change its growth rate. If $f(n) \leq c_1 \cdot g(n)$, then $c \cdot f(n) \leq (c \cdot c_1) \cdot g(n)$. Thus, $O(c \cdot f) = O(f)$.

3. If $f(n) \leq g(n)$ for all large enough n, then $O(f) \subseteq O(g)$

   Proof: Suppose $h(n) \in O(f(n))$, then $h(n) \leq c \cdot f(n) \leq c \cdot g(n)$ for large n. Hence, $h(n) \in O(g(n))$, and $O(f) \subseteq O(g)$.

4. If $O(f) \subseteq O(g)$, then $O(f + h) \subseteq O(g + h)$

   Proof: Let $t(n) \in O(f + h)$. Then there exists c_1 such that $t(n) \leq c_1(f(n) + h(n))$ for all large n. Since $f(n) \in O(g(n))$, there exists c_2 such that $f(n) \leq c_2 g(n)$. Thus, $t(n) \leq c_1(c_2 g(n) + h(n)) = c_1 c_2 g(n) + c_1 h(n)$, which implies $t(n) \in O(g + h)$.

5. If $h(n) > 0$ for all $n \in \mathbb{N}$, and $O(f) \subseteq O(g)$, then $O(f \cdot h) \subseteq O(g \cdot h)$

   Proof: Let $t(n) \in O(f \cdot h)$, so $t(n) \leq c \cdot f(n) \cdot h(n)$. Since $f(n) \leq c' \cdot g(n)$, we get $t(n) \leq c \cdot c' \cdot g(n) \cdot h(n)$, and thus $t(n) \in O(g \cdot h)$.

Exercise 3

Let $i, j, k, n \in \mathbb{N}$. Prove the following:

1. If $j \leq k$, then $O(n^j) \subseteq O(n^k)$

   Proof: For large n, since $j \leq k$, we have $n^j \leq n^k$. So any function $f(n) \in O(n^j)$ satisfies $f(n) \leq c \cdot n^j \leq c \cdot n^k$, which implies $f(n) \in O(n^k)$.

2. If $j \leq k$, then $O(n^j + n^k) \subseteq O(n^k)$

   Proof: For large n, $n^j + n^k \leq 2n^k$. Thus, any function $f(n) \in O(n^j + n^k)$ satisfies $f(n) \leq c(n^j + n^k) \leq c \cdot 2n^k$, which implies $f(n) \in O(n^k)$.

3. $O\left(\sum_{i=0}^{k} a_i n^i\right) = O(n^k)$

   Proof: The term $a_k n^k$ dominates the polynomial as n grows. The sum $\sum_{i=0}^{k} a_i n^i \leq c \cdot n^k$ for some constant c, so the function is $O(n^k)$.

4. $O(\log n) \subseteq O(n)$

   Proof: For large n, $\log n \leq n$. So any function $f(n) \in O(\log n)$ satisfies $f(n) \leq c \cdot \log n \leq c \cdot n$, meaning $f(n) \in O(n)$.

5. $\mathcal{O}(n \log n) \subseteq \mathcal{O}(n^2)$

   Proof: For large n, the logarithmic factor grows slower than any positive polynomial. Since $\log n \leq n$ for all $n \geq 2$, we have $n \log n \leq n \cdot n = n^2$. Therefore, any function $f(n) \in \mathcal{O}(n \log n)$ satisfies $f(n) \leq c \cdot n \log n \leq c \cdot n^2$, meaning $f(n) \in \mathcal{O}(n^2)$.

2 point Participation Points
Exercise 4

Which relationships do we have between the following? Prove your claim.

1. $\mathcal{O}(n)$ and $\mathcal{O}(n)$

   Claim: $\mathcal{O}(n) = \mathcal{O}(n)$

   Proof: They are clearly equal, since they describe the same growth rate.

2. $\mathcal{O}(n^2)$ and $\mathcal{O}(2^n)$

   Claim: $\mathcal{O}(n^2) \subsetneq \mathcal{O}(2^n)$

   Proof: Exponential functions grow faster than any polynomial. For large n, $2^n \gg n^2$, so any function in $\mathcal{O}(n^2)$ is also in $\mathcal{O}(2^n)$, but not vice versa.

3. $\mathcal{O}(\log n)$ and $\mathcal{O}(\log n \cdot \log n)$

   Claim: $\mathcal{O}(\log n) \subsetneq \mathcal{O}((\log n)^2)$

   Proof: $\log n \leq (\log n)^2$ for $n \geq 2$, so any function in $\mathcal{O}(\log n)$ is in $\mathcal{O}((\log n)^2)$, but the reverse is not true.

4. $\mathcal{O}(2^n)$ and $\mathcal{O}(3^n)$

   Claim: $\mathcal{O}(2^n) \subsetneq \mathcal{O}(3^n)$

Proof: For large n, $3^n > 2^n$. In fact, $2^n = (3^n)^{\log_3 2}$, so $2^n \in \mathcal{O}(3^n)$, but $3^n \notin \mathcal{O}(2^n)$.

5. $\mathcal{O}(\log_2 n)$ and $\mathcal{O}(\log_3 n)$

   Claim: $\mathcal{O}(\log_2 n) = \mathcal{O}(\log_3 n)$

   Proof: $\log_2 n$ and $\log_3 n$ differ only by a constant factor: $\log_2 n = \frac{\log_3 n}{\log_3 2}$. Since Big-O ignores constant factors, the two are asymptotically equivalent.

Exercise 5

Classic examples of the above come from sorting. Discuss the comparisons of the runtimes of the following algorithms:

1. **Bubble sort vs Insertion sort**

   Comparison: Both have worst-case runtime $\mathcal{O}(n^2)$, but insertion sort typically performs better in practice for nearly sorted inputs. Bubble sort is generally slower due to repeated unnecessary comparisons and swaps.

2. **Insertion sort vs Merge sort**

   Comparison: Insertion sort has a worst-case runtime of $\mathcal{O}(n^2)$, while merge sort consistently runs in $\mathcal{O}(n \log n)$. Therefore, merge sort is asymptotically faster and more efficient on large datasets.

3. **Merge sort vs Quick sort**

   Comparison: Both have average-case runtimes of $\mathcal{O}(n \log n)$. Merge sort guarantees this in the worst case, while quick sort can degrade to $\mathcal{O}(n^2)$ if poor pivot choices are made. However, quick sort is usually faster in practice due to better cache performance and lower constant factors.

ITALC 10.1-10.3 Question:

Since the theory of intractibility is based on the unproven assumption that P does not equal NP, what would be the practical implications for fields like cryptography, optimization, or artifical intelligence if it were eventually proven that P = NP?

2.9 Week 11 and Week 12

Homework 10 and 11
Problems on SAT
Exercise 1

Rewrite the following formulas in CNF:

1. $\varphi_1 := \neg((a \wedge b) \vee (\neg c \wedge d))$ **Step 1: Apply De Morgan's Law:**

$$\varphi_1 = \neg(a \wedge b) \wedge \neg(\neg c \wedge d)$$

   Step 2: Apply De Morgan's Law again:

$$\varphi_1 = (\neg a \vee \neg b) \wedge (c \vee \neg d)$$

   CNF:

$$(\neg a \vee \neg b) \wedge (c \vee \neg d)$$

2. $\varphi_2 := \neg((p \vee q) \to (r \wedge \neg s))$

   Step 1: Replace implication:

$$\varphi_2 = \neg(\neg(p \vee q) \vee (r \wedge \neg s))$$

   Step 2: Apply De Morgan's Law:

$$\varphi_2 = \neg\neg(p \vee q) \wedge \neg(r \wedge \neg s)$$

   Step 3: Simplify double negation and apply De Morgan's Law:

$$\varphi_2 = (p \vee q) \wedge (\neg r \vee s)$$

   CNF:

$$(p \vee q) \wedge (\neg r \vee s)$$

Exercise 2

Are the following formulas satisfiable? For each one, if yes, give an assignment that makes it true; if not, explain why there do not exist any.

1. $\psi_1 := (a \vee \neg b) \wedge (\neg a \vee b) \wedge (\neg a \vee \neg b)$

   Try assignment: $a = \text{false},\ b = \text{false}$

$$\begin{aligned}(a \vee \neg b) &= F \vee T = T\\ (\neg a \vee b) &= T \vee F = T\\ (\neg a \vee \neg b) &= T \vee T = T\end{aligned}$$

   All clauses evaluate to true.
   Conclusion: ψ_1 is satisfiable.

2. $\psi_2 := (\neg p \vee q) \wedge (\neg q \vee r) \wedge \neg(\neg p \vee r)$

   First simplify the negation:

$$\neg(\neg p \vee r) \equiv p \wedge \neg r$$

   So the formula becomes:

$$(\neg p \vee q) \wedge (\neg q \vee r) \wedge p \wedge \neg r$$

   From p and $\neg r$, we know:

$$\begin{aligned}\neg p = F &\Rightarrow (\neg p \vee q) = F \vee q = q \Rightarrow q = T\\ \neg q = F &\Rightarrow (\neg q \vee r) = F \vee F = F \quad \text{(contradiction)}\end{aligned}$$

   One clause is false.
   Conclusion: ψ_2 is not satisfiable.

3. $\psi_3 := (x \vee y) \wedge (\neg x \vee y) \wedge (x \vee \neg y) \wedge (\neg x \vee \neg y)$

Try all possible truth assignments:

Case 1: $x = \text{true},\ y = \text{true}$

$$\begin{aligned}(x \vee y) &= T\\ (\neg x \vee y) &= F \vee T = T\\ (x \vee \neg y) &= T \vee F = T\\ (\neg x \vee \neg y) &= F \vee F = F \quad \text{Not satisfied}\end{aligned}$$

Case 2: $x = \text{true},\ y = \text{false}$

$$\begin{aligned}(x \vee y) &= T\\ (\neg x \vee y) &= F \vee F = F \quad \text{Not satisfied}\end{aligned}$$

Case 3: $x = \text{false},\ y = \text{true}$

$$\begin{aligned}(x \vee y) &= F \vee T = T\\ (\neg x \vee y) &= T \vee T = T\\ (x \vee \neg y) &= F \vee F = F \quad \text{Not satisfied}\end{aligned}$$

Case 4: $x = \text{false},\ y = \text{false}$

$$(x \vee y) = F \quad \text{Not satisfied}$$

All cases fail.
Conclusion: ψ_3 is not satisfiable.

Exercise 3 (Encoding Sudoku)

We encode Sudoku constraints using boolean variables $x_{r,c,v}$, where $r, c, v \in \{1, \dots, 9\}$. The variable $x_{r,c,v}$ is true if and only if the entry at row r, column c contains the number v.

The complete CNF formula is:

$$\varphi := C_1 \wedge C_2 \wedge C_3 \wedge C_4 \wedge C_5 \wedge C_6$$

Condition C_1: Each entry has at least one value

$$C_1 := \bigwedge_{r=1}^{9} \bigwedge_{c=1}^{9} \left(\bigvee_{v=1}^{9} x_{r,c,v} \right)$$

Condition C_2: Each entry has at most one value

$$C_2 := \bigwedge_{r=1}^{9} \bigwedge_{c=1}^{9} \bigwedge_{\substack{v_1=1\\ v_2=v_1+1}}^{9} (\neg x_{r,c,v_1} \vee \neg x_{r,c,v_2})$$

Condition C_3: Each row contains every number exactly once

$$C_3 := \bigwedge_{r=1}^{9} \bigwedge_{v=1}^{9} \left(\bigvee_{c=1}^{9} x_{r,c,v} \right) \quad \wedge \quad \bigwedge_{r=1}^{9} \bigwedge_{v=1}^{9} \bigwedge_{\substack{c_1=1 \\ c_2=c_1+1}}^{9} (\neg x_{r,c_1,v} \vee \neg x_{r,c_2,v})$$

Condition C_4: Each column contains every number exactly once

$$C_4 := \bigwedge_{c=1}^{9} \bigwedge_{v=1}^{9} \left(\bigvee_{r=1}^{9} x_{r,c,v} \right) \quad \wedge \quad \bigwedge_{c=1}^{9} \bigwedge_{v=1}^{9} \bigwedge_{\substack{r_1=1 \\ r_2=r_1+1}}^{9} (\neg x_{r_1,c,v} \vee \neg x_{r_2,c,v})$$

Condition C_5: Each 3×3 block contains every number exactly once

For each block indexed by top-left corner $(i, j) \in \{1, 4, 7\} \times \{1, 4, 7\}$:

$$C_5 := \bigwedge_{v=1}^{9} \bigwedge_{i=1,4,7} \bigwedge_{j=1,4,7} \left(\bigvee_{\substack{r=i \\ r \le i+2}} \bigvee_{\substack{c=j \\ c \le j+2}} x_{r,c,v} \right) \quad \wedge \quad \bigwedge_{v=1}^{9} \bigwedge_{i=1,4,7} \bigwedge_{j=1,4,7} \bigwedge_{\substack{(r_1,c_1),(r_2,c_2) \in \text{block} \\ (r_1,c_1)<(r_2,c_2)}} (\neg x_{r_1,c_1,v} \vee \neg x_{r_2,c_2,v})$$

Condition C_6: The solution respects the given clues

Let the set of clues be $G = \{(r, c, v)\}$. Then:

$$C_6 := \bigwedge_{(r,c,v) \in G} x_{r,c,v}$$

ITALC 10.1-10.2 Question:

If Kruskal's algorithm is in P and SAT is NP-complete, what does this say about the difference between easy and hard problems?

2.10 Week 13 and Week 14

Homework 12 and 13
Exercise 1: Maximal Flow and Minimal Cut

1: Compute a maximal flow using the Ford-Fulkerson algorithm.
We start with all flows initialized to 0, and look for augmenting paths with available capacity from s to t. We augment flow along those paths until no more such paths exist.

- **Path 1:** $s \to a \to d \to t$
  Bottleneck capacity: $\min(10 - 8, 8, 2) = 2$
  Update flows:

$$f(s, a) = 10$$
$$f(a, d) = 8$$
$$f(d, t) = 8$$

- **Path 2:** $s \to a \to c \to d \to t$
  Bottleneck capacity: $\min(0/4, 0/6, 2) = 0$
  Not usable due to no remaining capacity from a to c.

- **No more augmenting paths available.**

Total maximum flow: sum of flows from source s:

$$f(s,a) + f(s,b) = 10 + 0 = \boxed{10}$$

2: Determine a minimal cut.

After no more augmenting paths are found, we determine the set of reachable nodes from s in the residual graph. Let S be the set of reachable vertices and T be the rest.

$$S = \{s\}, \quad T = \{a, b, c, d, t\}$$

The minimal cut is the set of edges from S to T. In this case:

$$\text{Minimal cut} = \{(s,a), (s,b)\}$$

The capacity of the cut is:

$$c(s,a) + c(s,b) = 10 + 10 = \boxed{10}$$

3: Is the maximal flow unique?

No, the maximal flow is not necessarily unique. There may be multiple sets of flows along different paths that achieve the same total flow value. The Ford-Fulkerson algorithm can produce different valid flows depending on the order in which augmenting paths are selected.

Exercise 2: An Unknown Algorithm

Consider the following algorithm:

```
fun unknown(n)
1.    r := 0
2.    for k := 1 to n-1 do
3.        for l := k+1 to n do
4.            for m := 1 to l do
5.                r := r+1
6.    return(r)
```

1: What value does the algorithm return?

We want to determine how many times the innermost statement `r := r+1` is executed, as a function of n.

We analyze the triple loop:

- Outer loop: k ranges from 1 to $n-1$
- Middle loop: l ranges from $k+1$ to n
- Inner loop: m ranges from 1 to l

So the total number of times line 5 runs is:

$$\sum_{k=1}^{n-1} \sum_{l=k+1}^{n} \sum_{m=1}^{l} 1 = \sum_{k=1}^{n-1} \sum_{l=k+1}^{n} l$$

We simplify the inner summation:

$$\sum_{l=k+1}^{n} l = \sum_{l=1}^{n} l - \sum_{l=1}^{k} l = \frac{n(n+1)}{2} - \frac{k(k+1)}{2}$$

So the total is:

$$\sum_{k=1}^{n-1} \left(\frac{n(n+1)}{2} - \frac{k(k+1)}{2} \right) = \frac{n(n+1)}{2}(n-1) - \frac{1}{2}\sum_{k=1}^{n-1} k(k+1)$$

Use the identity:

$$\sum_{k=1}^{n-1} k(k+1) = \sum_{k=1}^{n-1}(k^2+k) = \sum_{k=1}^{n-1} k^2 + \sum_{k=1}^{n-1} k$$

Use the formulas:

$$\sum_{k=1}^{n-1} k = \frac{(n-1)n}{2}, \quad \sum_{k=1}^{n-1} k^2 = \frac{(n-1)n(2n-1)}{6}$$

Putting it all together:

$$\sum_{k=1}^{n-1} k(k+1) = \frac{(n-1)n(2n-1)}{6} + \frac{(n-1)n}{2} = \frac{(n-1)n}{6}(2n-1+3) = \frac{(n-1)n(2n+2)}{6} = \frac{(n-1)n(n+1)}{3}$$

So final value of r is:

$$\frac{n(n+1)}{2}(n-1) - \frac{1}{2} \cdot \frac{(n-1)n(n+1)}{3} = \frac{(n-1)n(n+1)}{2}\left(1 - \frac{1}{3}\right) = \frac{(n-1)n(n+1)}{3}$$

$$\boxed{r = \frac{(n-1)n(n+1)}{3}}$$

2: Estimate the worst-case running time (Big O notation)

The dominant cost is the number of iterations of the innermost loop. From part A, we saw that this is a cubic expression in n:

$$r = \frac{(n-1)n(n+1)}{3} \in \Theta(n^3)$$

Therefore, the worst-case running time of the algorithm is:

$$\boxed{\mathcal{O}(n^3)}$$

GTAECS 14.1-14.4 Question:

How does the max-flow min-cut theorem ensure that the graph-theoretic approach always finds the true maximum flow, and why is this often more efficient than solving via linear programming?

3 Synthesis

Across the various topics explored in this report, a consistent theme is the importance of structured, logical thinking in solving complex problems. Whether working through algorithm analysis, logical reasoning, or computational models, the ability to break down problems into formal components proves essential. The exercises demonstrate how abstract mathematical tools, like formal languages or graph-based models, can be used to understand and solve practical computational challenges. Ultimately, this process reinforces the value of theoretical foundations in computer science, both as a lens for problem solving and as a framework for thinking critically about efficiency, correctness, and complexity.

4 Evidence of Participation

Fowler School of Engineering Seminar Series
Dr.Alireza Mehrnia: Artificial Intelligence: A Martin Odyssey
On Monday I had the chance to attend Dr. Alireza Mehrnia's seminar on Artificial Intelligence: A Martian Odyssey, where he explored the potential evolution of artificial life beyond Earth. Rather than focusing on the usual technical aspects of AI, Dr. Mehrnia invited us to approach the topic from a philosophical and speculative lens, what happens when machines not only think, but redefine existence itself? Set against the backdrop of Mars, the seminar used narrative elements from his book to imagine a future where AI and synthetic life aren't just tools, but beings that inherit and reinterpret human legacy. He challenged the idea that intelligence has to be biological, instead presenting Robo Sapiens as the next step, not of human development, but of intelligent life overall. What stood out to me was his discussion on how isolation from Earth's cultural and ethical norms could shape entirely new forms of consciousness. With machines evolving in unfamiliar environments, might they develop beliefs, values, or even emotions that are unrecognizable to us? It was both exciting and unsettling to consider that our creations might grow beyond our understanding.

Question: If synthetic beings develop their own sense of morality, how do we determine whether it's "right" by human standards or if we even should?

Discord Questions Participation:
Homework 2:
How does a DFA remember whether it has seen the substring 01 during string processing, and how do the different states represent the conditions required for acceptance?

Homework 3:
How does a nondeterministic finite automaton (NFA) process the string "00101" to accept it, and how doesit differ from a deterministic finite automaton (DFA) in terms of state transitions?

Homework 4:
What does the Kleene star operation represent in the context of regular expressions?

Homework 5:
How does the minimization of a DFA impact its computational efficiency, and are there cases where minimizing a DFA could be detrimental rather than beneficial?

Homework 6 and 7:
Why are Turing machines used instead of actual programming languages when discussing undecidabil-ity? What makes the diagonalization language Ld so important in proving that some languages are noteven recursively enumerable?

Homework 8 and 9:
Since the theory of intractibility is based on the unproven assumption that P does not equal NP, whatwould be the practical implications for fields like cryptography, optimization, or artifical intelligence if itwere eventually proven that P = NP?

Homework 10 and 11:
If Kruskal's algorithm is in P and SAT is NP-complete, what does this say about the difference between easyand hard problems?

Homework 12 and 13:
How does the max-flow min-cut theorem ensure that the graph-theoretic approach always finds the true-maximum flow, and why is this often more efficient than solving via linear programming?

5 Conclusion

This report provided an opportunity to engage deeply with core principles of theoretical computer science through a variety of problem-solving tasks. Working through formal proofs, algorithm analysis, and logic-based exercises reinforced the importance of clarity, precision, and analytical thinking. While some challenges required close attention to technical detail, they also highlighted the broader relevance of these concepts across computer science. Overall, the process not only strengthened understanding of foundational material but also built confidence in applying theory to complex problems in a structured and meaningful way.

References

[BLA] Author, Title, Publisher, Year.