CPSC-406 Report

Jaime Song
Chapman University

02/16/25

Abstract

Contents

1 Introduction

2 Week by Week

2.1 Week 1

Lab 1 Introduction to Automata Theory
Homework Question:
Characterize all the accepted words (i.e., describe exactly those words that get recognized).
Answer:

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

```
5a + 10b = 25
a = num of times 5 appears
b = num of times 10 appears
a,b = non-negative integers
valid values of (a,b) are:
(5,0) = 5,5,5,5,5
(3,1) = 5,5,5,10
(1,2) = 5,10,10
```

Homework Question:
Characterize all the accepted words. Can you describe them via a regular expression?
Answer:

```
A word is accepted if it consists of one or more "pay" actions followed by a "push" and this
    sequence can repeat
any number of times
Expression:
(pay + push)+
```

2.2 Week 2

Lab 1 Deterministic and Non-deterministic Finite Automata (DFAs and NFAs)
Homework Question:
Determine for the following words, if they are contained in L1, L2, or L3
Answer:

```
w1 = 10011
  L1 = yes
  L2 = no
  L3 = no
w2 = 100
  L1 = no
  L2 = no
  L3 = no
w3 = 10100100
  L1 = yes
  L2 = yes
  L3 = yes
w4 = 1010011100
  L1 = yes
  L2 = no
  L3 = no
w5 = 11110000
  L1 = no
  L2 = yes
  L3 = yes
```

Homework Question:
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?
Answer:

```
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 3

Lab 2
Programming (with) Automata
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

```
1. 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
```

2.

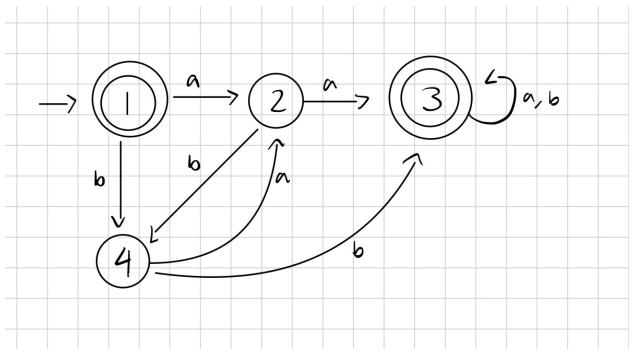

Figure 1:

```
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 4

Homework 3
Operations on Automata
HW 1 (Extended Transition Function):

```
1. The automataon A^2 accepts all strings that contain an odd number of As. If a string has an
    even number of As (including 0) it is rejected

2. A^1:
(1,a) -> 2
(1,b) -> 4
(1,a) -> 4
(1,a) -> 4

(1,abaa) = 4

A^2:
(1,a) -> 2
(2,b) -> 1
(1,b) -> 3
(3,a) -> 3

(1,abba) = 3 (not final state so not accepted)
```

HW 2 (Product Automata:)
1.

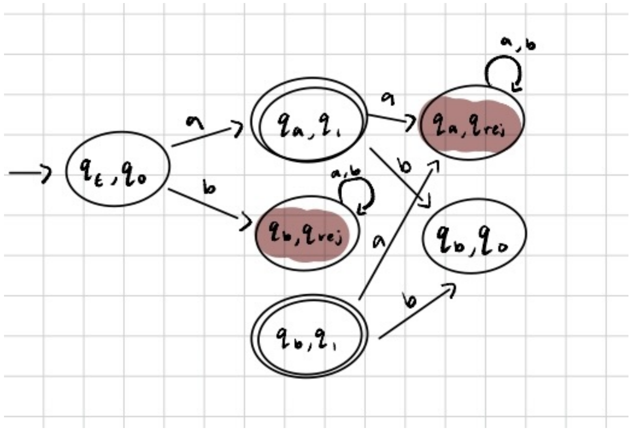

Figure 2:

```
\textbf{HW 2 (Product Automata:)}
\begin{lstlisting}
1.

2. A string is accpeted by A if and only if it is accepted by both A^1 and A^2
* By definition, A tracks the simultaneous behavior of A^1 and A^2
* A string w is accepted in A if and only if both automata reach a final state after processing w
* Since A is constructed with states (q^1, q^2), the final states of A are precisely those where
    both components are final
```

```
* This proves the equation

3. To change A to recognize L(A^2) U L(A^2) we can modify its final states
* Rather than requiring both automata to be in final state, we can accept if either automaton is
    in a final state
* change from:
F = F^2 x F^2
to:
F' = (F^1 x Q^2) U (Q^1 x F^2)

* means string is accepted by either A^1 or A^2
```

ITALC: Exercise 2.2.7

```
Proof:
Let A be a DFA, q a state of A such that \delta(q,a) = q for all a. We will prove by induction on
    the length of the input string w that d(q,w) = q for all w.

Base Case:
For w = \epsilon, we have d(q,\epsilon) = q by definition, so the base case holds.

Inductive Step:
Assume d(q,w) = q for some string w of length n. For w' = wa, we have:
d(q,w\prime) = \delta(d(q,w),a) = \delta(q,a) = q
Thus, d(q,w\prime) = q, and the inductive step holds.

By induction, d(q,w) = q for all w.
```

ITALC: Section 2.3

```
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 5

Homework 4
Determinization of NFAs
HW 1:

```
1. A DFA is a special case of an NFA. In an NFA, the transition function allows multiple possible
    states for a given input, while in the DFA, the transition function maps each state and input
    to a single state.
Since every DFA is inherently an NFA where the transition function always returns a unique next
    state, we can view A as an NFA without making any modifications. We reinterpret the DFA as an
    NFA where:
* The state set remains the same
* The transition function is modified to return singleton sets
* The start state and accepting state remain unchanged

2. Given a general DFA A = (Q, sigma, delta, q0, F) we can define an equivalent NFA A' = (Q',
    sigma', delta', q0', F') where:
Q' = Q
sigma remains the same
delta': Q' x sigma -> P(Q') is defined as:
```

```
delta'(q,a) = {delta(q,a)} for all q belongs to Q, a belongs to sigma
This ensures that each transition in A is preserved but is now viewed as returning a set of states
    rather than a single state
q'0 = q0 (initial state remains unchanged)
F' = F (final states remain unchanged)
```

HW 2:

```
1. Looking at the NFA:
* It starts at q0, where it loops on both 0 and 1
* A transition to q1, happens on input 1
* From q1, it can go to q2 on 0 or back to itself on 1
* From q2, it transitions to q3 on 0
* q3 is a final state and loops on both 0 and 1
The NFA recognizes the language of binary strings that contain 100 as a substring. The accepting
    state q3 is reached whenever the sequence 100 appears, and it stays in q3 accepting all
    further inputs

2. NFA A given by:
* Q = {q0, q1, q2, a3}
* Sigma = {0,1}
* Transition function delta:
delta(q0, 0) = {q0}
delta(q0, 1) = {q0,q1}
delta(q1, 0) = {q2}
delta(q1, 1) = {q1}
delta(q2, 0) = {q3}
delta(q2, 1) = {q1}
delta(q3, 0) = {q3}
delta(q3, 1) = {q3}
* Start state: q0
* Accepting states: F = {q3}

3. Step by step
  1. delta^(q0,1) = {q0,q1} (since q0 transitions to both itself and q1 on 1)
  2. delta^({q0, q1}, 0) = {q0, q2} (since q0 loops on 0 and q1 moves to q2)
  3. delta^({q0, q2}, 1) = {q0, q1} (since q0 loops on 1 and q2 transitions to q1 on 1)
  4. delta^({q0, q1}, 1) = {q0, q1} (same reason as step 1)
  5. delta^({q0, q1}, 0) = {q0, q2} (same reason as step 2)
Final states: {q0,q2} which does not include q3, so 101110 is rejected
```

4 and 5.

```
6. {q0,q2,q3} and {q0,q1,q3} can be combined into one state
```

ITALC: Section 3.1, 3.2.1, 3.2.2

```
Question: What does the Kleene star operation represent in the context of regular expressions?
```

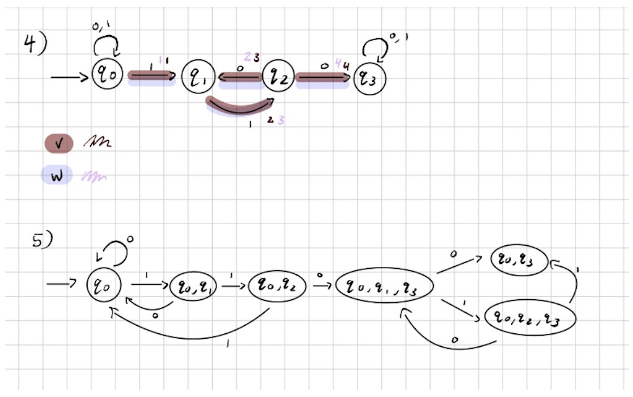

Figure 3:

2.6 Week 6

Homework 5 ITALC 3.2
Section 3.2.1
a. Here, $R_{ij}^{(0)}$ represents the regular expression for transitioning from state q_i to state q_j using exactly zero states as intermediates. These are simply based on the direct transitions:

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

b. Now we calculate the regular expressions using one intermediate state (state q_1).

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

c. Now we calculate the regular expressions using up to two intermediate states (q_1 and q_2).

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

d. The language of the automaton is defined by all possible strings leading to the final state q_3.

We observe: - q_3 is the only accepting state. - The simplest regular expression can be derived by considering all transitions leading to q_3. - The language consists of strings ending in 00 or 000, as they lead to the final state.

Thus, the regular expression for the language is:

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

Section 3.2.1
c. Using up to two intermediate states (q_1 and q_2), we calculate the regular expressions for all state pairs:

$$\begin{aligned}
R_{11}^{(2)} &= R_{11}^{(1)} \cup (R_{12}^{(1)} R_{21}^{(1)}) = \epsilon \cup (0 \cdot 1) = \epsilon \cup 01 \\
R_{12}^{(2)} &= R_{12}^{(1)} \cup (R_{12}^{(1)} R_{22}^{(1)}) = 0 \cup (0 \cdot \epsilon) = 0 \\
R_{13}^{(2)} &= R_{13}^{(1)} \cup (R_{12}^{(1)} R_{23}^{(1)}) = 1 \cup (0 \cdot 0) = 1 \cup 00 \\
R_{21}^{(2)} &= R_{21}^{(1)} \cup (R_{22}^{(1)} R_{21}^{(1)}) = 1 \cup (\epsilon \cdot 1) = 1 \cup 1 = 1 \\
R_{22}^{(2)} &= R_{22}^{(1)} \cup (R_{22}^{(1)} R_{22}^{(1)}) = \epsilon \cup (\epsilon \cdot \epsilon) = \epsilon \\
R_{23}^{(2)} &= R_{23}^{(1)} \cup (R_{22}^{(1)} R_{23}^{(1)}) = 0 \cup (\epsilon \cdot 0) = 0 \\
R_{31}^{(2)} &= R_{31}^{(1)} \cup (R_{32}^{(1)} R_{21}^{(1)}) = 1 \cup (1 \cdot 1) = 1 \cup 11 \\
R_{32}^{(2)} &= R_{32}^{(1)} \cup (R_{32}^{(1)} R_{22}^{(1)}) = 1 \cup (1 \cdot \epsilon) = 1 \\
R_{33}^{(2)} &= R_{33}^{(1)} \cup (R_{32}^{(1)} R_{23}^{(1)}) = \epsilon \cup (1 \cdot 0) = \epsilon \cup 10
\end{aligned}$$

d. The language of the DFA consists of all possible strings that lead to the accepting state q_3. From the table, we observe that:

- State q_3 is the accepting state. - q_3 can be reached directly using 1 or through sequences like 00. - Using the repeating structure, the regular expression for the language can be written as:

$$1^*(00)^*1^*$$

This expression represents all valid sequences accepted by the DFA.

ITALC 4.4 — Section 4.4.1 a.

	q_1	q_2	q_3
q_1			
q_2			
q_3			

1. **Mark accepting and non-accepting pairs**: - (q_1, q_3) and (q_2, q_3) since q_3 is accepting.

2. **Apply transitions**: - On input **0** and **1**, follow state transitions and check if they lead to distinguishable pairs. - Update the table accordingly.

3. **Final table**:

	q_1	q_2	q_3
q_1	ϵ	Indistinguishable	Distinguishable
q_2	Indistinguishable	ϵ	Distinguishable
q_3	Distinguishable	Distinguishable	ϵ

b. - **q_1 and q_2 are equivalent** because they are not distinguishable. - q_3 remains separate as it is distinguishable from all others.

Thus, the reduced DFA has: - **Two states**: $[q_1, q_2]$ and 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

Section 4.4.2
a.

	A	B	C	D
A	ϵ			
B		ϵ		
C			ϵ	
D				ϵ

1. **Mark accepting and non-accepting pairs**: - If one state is accepting and the other is not, mark them as distinguishable.

2. **Apply transitions**: - Check where states transition on inputs **0** and **1**. - If transitions lead to distinguishable pairs, update the table.

3. **Final table after refinement**:

	A	B	C	D
A	ϵ	Indistinguishable	Distinguishable	Distinguishable
B	Indistinguishable	ϵ	Distinguishable	Indistinguishable
C	Distinguishable	Distinguishable	ϵ	Distinguishable
D	Distinguishable	Indistinguishable	Distinguishable	ϵ

b. - **C, F, I are equivalent** because they are all accepting states and indistinguishable. - **D and G are equivalent**, as they have the same transitions. - **E and H are equivalent**, as they behave identically.

Thus, the reduced DFA has: - **Five states**: $[A]$, $[B]$, $[C, F, I]$, $[D, G]$, $[E, H]$. - **Transitions**: - $[A] \xrightarrow{0} [B]$ - $[A] \xrightarrow{1} [E, H]$ - $[B] \xrightarrow{0} [C, F, I]$ - $[B] \xrightarrow{1} [E, H]$ - $[C, F, I] \xrightarrow{0} [D, G]$ - $[C, F, I] \xrightarrow{1} [H, E]$ - $[D, G] \xrightarrow{0} [E, H]$ - $[D, G] \xrightarrow{1} [C, F, I]$ - $[E, H] \xrightarrow{0} [A]$ - $[E, H] \xrightarrow{1} [C, F, I]$ - **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 11

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 Sections: 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 12

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 Sections 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 13

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 $r, c, v \in \{1, \ldots, 9\}$. 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 Sections 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 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

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?

5 Conclusion

References

[BLA] Author, Title, Publisher, Year.