Introduction to Automata Theory
Abstract Machines and Formal Languages

Max Randall
Chapman University

May 18, 2025

Abstract

Automata are mathematical models of computation capturing devices that operate without unbounded memory. In this report we introduce the basic definitions of deterministic and nondeterministic finite automata through illustrative examples, and outline their role in defining and recognizing regular languages.

Contents

1 Week 1: Finite Automata Fundamentals

1.1 Introduction

Automata are abstract machines that model computations without memory. Before defining them formally, we consider some examples.

1.2 Readings

Parking or Vending Machine

Specification: The machine requires 25 cents, paid in chunks of 5 or 10 cents.

Automaton: The state 25 is the accepting or final state. A word (i.e., a sequence of symbols 5 and 10) is accepted if it leads from the initial state (0) to the final state (25).

$$\begin{tikzcd}
\text{start} \arrow[r] & \text{0¢} \arrow[r, bend left, "\text{5¢}"] \arrow[rr, bend left=50, "\text{10¢}"] & \text{5¢} \arrow[r, bend left, "\text{5¢}"] \arrow[rr, bend left=50, "\text{10¢}"] & \text{10¢} \arrow[r, bend left, "\text{5¢}"] \arrow[rr, bend left=50, "\text{10¢}"] & \text{15¢} \arrow[r, bend left, "\text{5¢}"] \arrow[rr, bend left=50, "\text{10¢}"] & \text{20¢} \arrow[r, bend left, "\text{5¢}"] & \text{25¢}
\end{tikzcd}$$

Variable Names

Specification: In defining a programming language, valid variable names should:

- Start with a letter ($\ell = a, b, c, \ldots, z$).
- Be followed by any combination of letters (ℓ) or digits ($d = 0, 1, 2, \ldots, 9$).
- End with a terminal symbol (t, e.g., $t =;$).

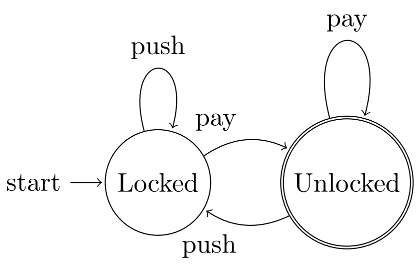
Automaton: Accepted words follow the pattern:

$$\ell(\ell + d)^*t$$

Turnstile

Specification: A money-operated turnstile:

- Starts in the **locked** state.
- From **locked**:
  - A **push** (u) keeps it **locked**.
  - A **pay** (p) moves it to the **unlocked** state.
- From **unlocked**:
  - A **pay** (p) keeps it **unlocked**.
  - A **push** (u) moves it back to **locked**.
- The **unlocked** state is the accepting state.



1.3 Homework

Exercise: Characterizing Accepted Words

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

The vending machine automaton accepts words consisting of payments in increments of 5 and 10 cents, reaching exactly 25 cents. The accepted words follow the pattern:

$$(10 + 5)^*$$

subject to the constraint that the sum of the numbers in the sequence equals 25.

Exercise: Turnstile Regular Expression

Characterize all accepted words and describe them using a regular expression.

The turnstile automaton can be characterized by the following regular expression:

$$(p + up)^* p (p + up)^*$$

where:

- p represents a **pay** action.
- u represents a **push** action.
- p^* means zero or more additional **pay** actions while the turnstile remains open.
- The entire sequence can repeat any number of times.

Example Accepted Words

p	(Pay once, unlocks, and stays open)
pp	(Pay twice, remains unlocked)
pup	(Pay, push to lock, then pay again to unlock)
$ppuppp$	(Multiple pays, a push to lock, then more pays)
$upupup$	(Invalid pushes in locked state, followed by pays to unlock)

Exercise: Word Classification in Languages

Determine for the following words if they are contained in L_1, L_2, or L_3.

Here, based on the descriptions given later in the report:

- L_1 is the set of words that contain the substring "01".
- L_2 is the set of words whose lengths are powers of two.
- L_3 is the set of words with an equal number of 0s and 1s.

The words under consideration are:

	L_1	L_2	L_3
$w_1 = 10011$	Yes		
$w_2 = 100$			
$w_3 = 10100100$	Yes	Yes	
$w_4 = 1010011100$	Yes		Yes
$w_5 = 11110000$		Yes	Yes

Explanation:

- $w_1 = 10011$: Contains the substring "01" (specifically, the third and fourth symbols form "01"). Its length is 5 (not a power of two), and it has 3 ones versus 2 zeros.
- $w_2 = 100$: Does not contain the substring "01" (the pairs are "10" and "00"), its length is 3 (not a power of two), and it has 1 one and 2 zeros.
- $w_3 = 10100100$: Contains "01" (for example, the second and third symbols form "01"); its length is 8 (which is 2^3); however, it has 3 ones and 5 zeros.
- $w_4 = 1010011100$: Contains "01" (e.g., the first occurrence between the first and second symbols or elsewhere); its length is 10 (not a power of two); and it has 5 ones and 5 zeros.
- $w_5 = 11110000$: Does not contain the substring "01" (the transition from 1s to 0s gives "10" rather than "01"); its length is 8 (a power of two); and it has 4 ones and 4 zeros.

Exercise: DFA Run Acceptance

Consider the DFA from above (see dfa_example). Consider the paths corresponding to the words $w_1 = 0010$, $w_2 = 1101$, and $w_3 = 1100$. For which of these words does their run end in the accepting state?

Definition. We call

$$L(\mathcal{A}) := \{w \in \Sigma^* \mid \text{The run for } w \text{ in } \mathcal{A} \text{ ends in some } q \in F\}$$

the language *accepted* by $\mathcal{A}$.

Answer: After tracing the transitions in the given DFA, we find that:

- For $w_1 = 0010$, the run ends in a non-accepting state.
- For $w_2 = 1101$, the run ends in a non-accepting state.
- For $w_3 = 1100$, the run ends in an accepting state.

One plausible interpretation (consistent with common examples such as a DFA for determining divisibility by 3 or for ensuring an even number of 1s) shows that only w_3 meets the acceptance condition, while w_1 and w_2 do not.

Summary of Week 1

Finite automata are abstract machines used to recognize regular languages, which can be fully described using finite-state transitions. This chapter explores deterministic finite automata (DFAs) and nondeterministic finite automata (NFAs), demonstrating that NFAs, despite their flexibility, recognize the same class of languages as DFAs.

A key distinction between the two is that DFAs have a single active state at any time, whereas NFAs may simultaneously exist in multiple states. While NFAs simplify language representation, they can always be converted into equivalent DFAs through an algorithmic transformation.

An important extension of NFAs allows transitions on the empty string, further enhancing their expressiveness while still recognizing only regular languages. These ε-NFAs will later play a crucial role in proving the equivalence of finite automata and regular expressions.

The chapter also introduces an applied perspective on finite automata through a real-world example: electronic money protocols. By modeling interactions between a bank, a store, and a customer as finite automata, the protocol's correctness and potential fraud vulnerabilities can be analyzed. The study concludes with constructing a product automaton to validate interactions, ensuring that transactions occur as intended while preventing unauthorized duplication or cancellation of funds.

1.4 Conclusion

Deterministic Finite Automata (DFAs) are a fundamental concept in automata theory, providing a mathematical model for recognizing patterns in strings. A DFA is defined as a tuple $\mathcal{A} = (Q, \Sigma, \delta, q_0, F)$, where Q is a finite set of states, Σ is an input alphabet, δ is the transition function mapping states and symbols to new states, q_0 is the initial state, and F is the set of accepting states. DFAs process input strings deterministically, meaning that for every state and input symbol, there is exactly one transition.

Formal languages are sets of words over an alphabet Σ. The set of all possible words is denoted Σ^*, which includes the empty word ε. The length of a word w is written as $|w|$, and the occurrence of a symbol a in w is denoted $|w|_a$. Example languages include L_1, which contains words with the substring "01," L_2, which consists of words whose lengths are powers of two, and L_3, which contains words with an equal number of 0s and 1s.

DFAs determine if a word belongs to a language by processing transitions. If the final state is in F, the word is accepted; otherwise, it is rejected.

2 Week 2: DFA Implementation in Python

2.1 Introduction

In this homework, we explore the implementation of deterministic finite automata (DFAs) in Python. We will go beyond the simple graphical representation of DFAs and build them using Python types.

2.2 Readings

Chapter 2.2.4

A **Deterministic Finite Automaton (DFA)** is a formal model of computation that processes input sequences while maintaining a single, well-defined state at any given time. The term "deterministic" means that for each input symbol, the automaton transitions to exactly one state.

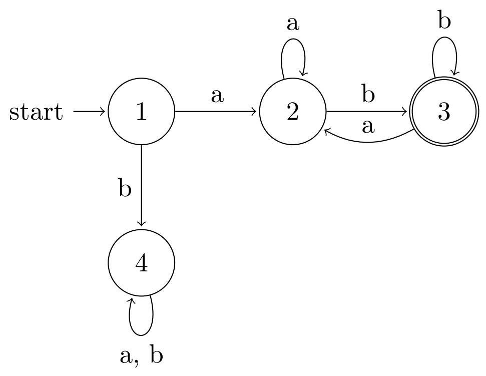
2.3 Homework

Implementing DFAs in Python

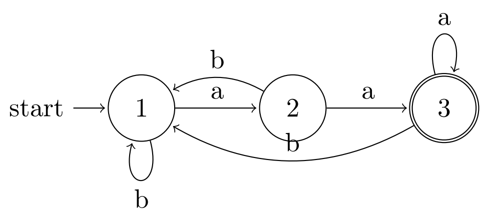
We will implement the following DFAs in Python using this `DFA` class:

```
class DFA:

    def __init__(self, Q, Sigma, delta, q0, F):
        self.Q = Q  # Set of states
        self.Sigma = Sigma  # Alphabet
        self.delta = delta  # Transition function (dict)
        self.q0 = q0  # Initial state
        self.F = F  # Set of accepting states

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

    def run(self, w):
        current_state = self.q0  # Start at initial state
        loop = 0
        for symbol in w:
            loop += 1
            if symbol not in self.Sigma:
                return False  # Reject if symbol is not in the alphabet
            if (current_state, symbol) not in self.delta:
                return False  # Reject if there's no valid transition
            current_state = self.delta[(current_state, symbol)]  # Move to next state

        return current_state in self.F  # Accept if final state is in F
```

Exercise 1: Word Processing with DFAs



Automaton $\mathcal{A}_1$



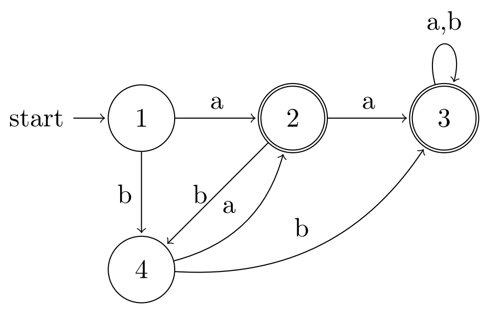
Automaton $\mathcal{A}_2$

Here is how we initialize the DFAs in Python:

```
# DFA A1
Q = {1,2,3,4}
Sigma = {'a','b'}
delta = {(1,'a'):2, (1,'b'):4, (2,'a'):2, (2,'b'):3,
         (3,'a'):2, (3,'b'):2, (4,'a'):4, (4,'b'):4}
q0 = 1
F = {3}
A1 = dfa.DFA(Q, Sigma, delta, q0, F)

# DFA A2
Q = {1,2,3}
Sigma = {'a','b'}
delta = {(1,'a'):2, (1,'b'):1, (2,'a'):3, (2,'b'):1,
         (3,'a'):3, (3,'b'):1}
q0 = 1
F = {3}
A2 = dfa.DFA(Q, Sigma, delta, q0, F)
```

Constructing the Complement DFA

To construct an automaton A_0 that accepts exactly the words that A refuses and vice versa we can simply swap the accepting states with the previously non-accepting states.

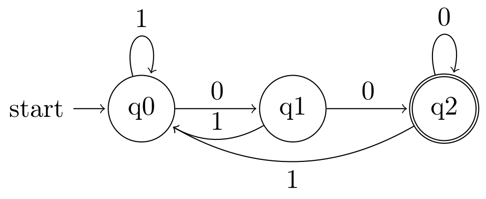


We can represent this operation with a `refuse()` function in Python:

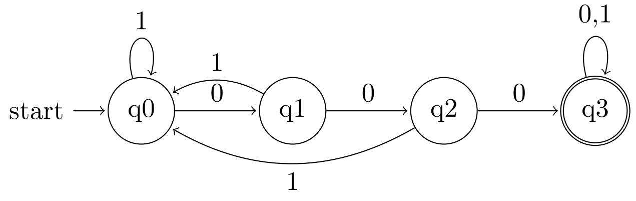
```
def refuse(A):
    """Constructs a DFA A0 that accepts exactly the words that A refuses and vice versa."""
    Q0 = A.Q
    Sigma0 = A.Sigma
    delta0 = A.delta
    q0_0 = A.q0
    F0 = Q0 - A.F  # Complement of the accepting states

    return dfa.DFA(Q0, Sigma0, delta0, q0_0, F0)
```

Exercise 2.4.4

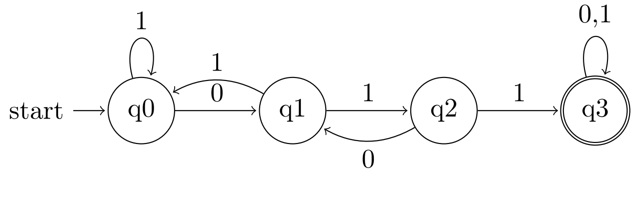
(a) DFA for strings ending in 00



(b) DFA for strings containing 000 as a substring



(c) DFA for strings containing 011 as a substring



2.4 Conclusion

A **Deterministic Finite Automaton (DFA)** is a theoretical computational model used to recognize formal languages. A DFA consists of a finite set of states Q, an input alphabet Σ, a transition function $\delta : Q \times \Sigma \to Q$, a start state q_0, and a set of accepting states F. The machine processes strings sequentially, transitioning between states according to δ. If, after consuming the entire string, the DFA ends in an accepting state, the input is accepted; otherwise, it is rejected.

3 Week 3: DFAs and NFAs

3.1 Introduction

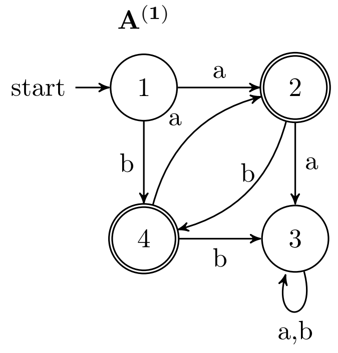
This week we covered ways to combine automata. We focused on the set theory used in the combination as well as some notation. Lastly, we introduced Nondeterministic Finite Automatas, or NFAs.

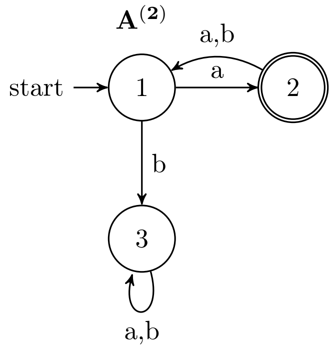
3.2 Homework

Extended Transition Functions

Brief summary of the question: Given the two DFAs below:

- Compute the extended transition functions $\hat{\delta}^{(1)}(1,\, abaa)$ and $\hat{\delta}^{(2)}(1,\, abba)$, showing all steps.
- Describe the language accepted by each automaton.





$$L\big(A^{(1)}\big) = \{\, w \in \{a,b\}^{+} \mid \text{no two consecutive symbols are the same}\},$$

$$L\big(A^{(2)}\big) = a\,((a \mid b)\,a)^{*} = \{\, w \in \{a,b\}^{*} \mid w \text{ has odd length, starts with } a\text{, and every odd position is } a\}.$$

$$\hat{\delta}^{(1)}(1,\, abaa) = 3, \quad \hat{\delta}^{(2)}(1,\, abba) = 3.$$

Intersection Automaton for $A^{(1)}$ and $A^{(2)}$

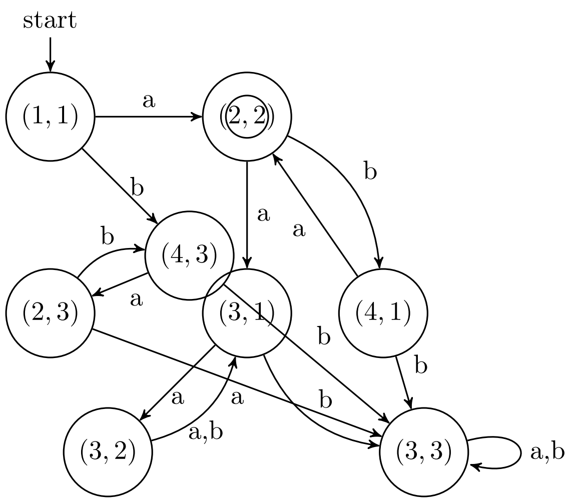
We form the product automaton

$$A = (Q^{(1)} \times Q^{(2)},\, \Sigma,\, \delta,\, (q_0^{(1)}, q_0^{(2)}),\, F^{(1)} \times F^{(2)}),$$

where

$$Q^{(1)} = \{1,2,3,4\},\ Q^{(2)} = \{1,2,3\},\ \Sigma = \{a,b\},$$

$\delta((p,q),x) = (\delta^{(1)}(p,x), \delta^{(2)}(q,x))$, and accepting states $F^{(1)} \times F^{(2)}$.



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

A simulates both in parallel and accepts iff both coordinates are accepting.

Changing to Obtain Union

Use the same transitions but set

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

so $L(A') = L(A^{(1)}) \cup L(A^{(2)})$.

Exercise 2.2.7

Claim. If $\delta(q, a) = q$ for all a, then $\delta(q, w) = q$ for any w.

Proof (by induction): Base: $\delta(q, \varepsilon) = q$. Step: $\delta(q, xa) = \delta(\delta(q, x), a) = \delta(q, a) = q$.

3.3 Readings

Chapter 2.3

The extended transition function for an NFA,

$$\hat{\delta}(q, \varepsilon) = \{q\}, \quad \hat{\delta}(q,\, xa) = \bigcup_{p \in \hat{\delta}(q,x)} \delta(p, a),$$

captures nondeterminism. The subset construction converts an NFA N into a DFA D whose states are subsets of N's states, preserving $L(D) = L(N)$.

3.4 Conclusion

Throughout these problems and readings, we deepened our understanding of DFAs and NFAs: extended transition functions, intersection/union constructions, and the subset construction proving NFA–DFA equivalence.

Interesting question: Maximal Blow-Up in the Subset Construction.
Describe an n-state NFA forcing all 2^n subsets to appear in its equivalent DFA and prove minimality.

4 Week 4: Determinization

4.1 Introduction

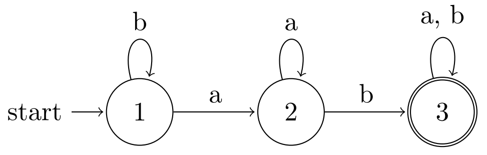
In this report, we explore fundamental aspects of both deterministic and nondeterministic finite automata (DFAs and NFAs), including extended transition functions, product automaton construction for intersection, and state modifications for union. We further demonstrate the subset construction to establish the equivalence of NFAs and DFAs, showcasing the broad utility of these automata concepts in both theoretical and practical contexts.

4.2 Homework

Homework 1

Let $\mathcal{A} = (Q, \Sigma, \delta : Q \times \Sigma \to Q, q_0, F)$ be a DFA. Explain in what way you can view $\mathcal{A}$ as an NFA by doing the following:

1. Let $\mathcal{A}$ denote the following DFA:



   Here:

$$
\begin{aligned}
Q &= \{1,2,3\}, \quad \Sigma = \{a,b\}, \\
q_0 &= 1, \quad F = \{3\}, \\
\delta(1,a) &= 2,\ \delta(1,b) = 1, \\
\delta(2,a) &= 2,\ \delta(2,b) = 3, \\
\delta(3,a) &= 3,\ \delta(3,b) = 3.
\end{aligned}
$$

   Explain how you can understand $\mathcal{A}$ also as an NFA.

2. More generally, let $\mathcal{A} = (Q, \Sigma, \delta : Q \times \Sigma \to Q, q_0, F)$ be a DFA. Define an NFA

$$\mathcal{A}' = (Q', \Sigma, \delta' : Q' \times \Sigma \to \mathcal{P}(Q'), q_0', F')$$

   such that $L(\mathcal{A}) = L(\mathcal{A}')$.

3. Justify why your construction satisfies the desired condition.

Solutions

1) Viewing the Example DFA as an NFA A deterministic finite automaton (DFA) can be seen as a special case of a nondeterministic finite automaton (NFA) by interpreting its transition function in the following way: in an NFA, the transition function

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

produces *sets* of possible next states. In a DFA, however, for each state q and input symbol a, there is exactly one next state $\delta(q, a)$. To view the DFA as an NFA, we simply set:

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

Hence, each transition in the original DFA becomes a transition to a *singleton set* in the NFA, preserving the recognized language because no extra nondeterminism is introduced.

2) General Construction from a DFA to an NFA Given any DFA $\mathcal{A} = (Q, \Sigma, \delta, q_0, F)$, define an NFA $\mathcal{A}' = (Q', \Sigma, \delta', q_0', F')$ by:

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

3) Why $L(\mathcal{A}) = L(\mathcal{A}')$? Because each step in the DFA corresponds to a unique singleton transition in the NFA:

- If a string w is accepted by the DFA, the identical path exists in the NFA.
- If a string w is rejected by the DFA, no path in the NFA leads to acceptance.

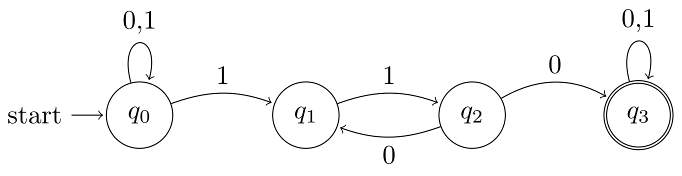
Thus the two automata recognize the same language.

Homework 2

1. Describe in words the language $L(\mathcal{A})$ accepted by the NFA $\mathcal{A}$ pictured below.
2. Specify the automaton $\mathcal{A}$ formally as $(Q, \Sigma, \delta, q_0, F)$.
3. Using the extended transition function $\hat{\delta}$, compute $\hat{\delta}(q_0, 10110)$ step by step.
4. Find *all* paths in $\mathcal{A}$ for $v = 1100$ and $w = 1010$; represent each set of paths.
5. Construct the determinization $\mathcal{A}^D$ (power-set automaton) of $\mathcal{A}$.
6. Verify $L(\mathcal{A}) = L(\mathcal{A}^D)$. Is there a smaller DFA for the same language?

Solutions

Below is the NFA $\mathcal{A}$:



1) Language Description $\mathcal{A}$ accepts exactly those binary strings that can nondeterministically traverse $q_0 \to q_1 \to q_2$ on two 1's and then take a 0 to q_3, after which any symbols are allowed.

2) Formal Specification

$$\mathcal{A} = (Q, \Sigma, \delta, q_0, F), \quad Q = \{q_0, q_1, q_2, q_3\},\ \Sigma = \{0, 1\},\ F = \{q_3\},$$

with

$$\begin{aligned} &\delta(q_0, 0) = \{q_0\},\ \delta(q_0, 1) = \{q_0, q_1\}, \\ &\delta(q_1, 1) = \{q_2\},\ \delta(q_1, 0) = \varnothing, \\ &\delta(q_2, 0) = \{q_1, q_3\},\ \delta(q_2, 1) = \varnothing, \\ &\delta(q_3, a) = \{q_3\} \quad (a \in \{0, 1\}). \end{aligned}$$

3) Extended Transition on 10110 $\hat{\delta}(q_0, \varepsilon) = \{q_0\}$, $\hat{\delta}(q_0, 1) = \{q_0, q_1\}$, $\hat{\delta}(\{q_0, q_1\}, 0) = \{q_0\}$, $\hat{\delta}(\{q_0\}, 1) = \{q_0, q_1\}$, $\hat{\delta}(\{q_0, q_1\}, 1) = \{q_0, q_1, q_2\}$.

4) All Paths for 1100 and 1010 Branches arise at each nondeterministic choice; any accepting path must include $q_0 \xrightarrow{1} q_1 \xrightarrow{1} q_2 \xrightarrow{0} q_3$.

5) Determinization (Power-Set)

$$Q^D = \mathcal{P}(Q),\ q_0^D = \{q_0\},\ F^D = \{S : S \cap \{q_3\} \neq \varnothing\},\ \delta^D(S, a) = \bigcup_{s \in S} \delta(s, a).$$

6) Verification & Minimization By construction $L(\mathcal{A}^D) = L(\mathcal{A})$. The resulting DFA can then be minimized via standard algorithms.

4.3 Conclusion

By viewing any DFA as an NFA with singleton transitions and applying the subset (powerset) construction, we have shown that NFAs and DFAs recognize exactly the same class of languages. This determinization process not only establishes the theoretical equivalence of nondeterministic and deterministic finite automata but also provides a concrete algorithm for converting an NFA into an equivalent DFA, reinforcing the practical utility of these foundational automata concepts.

5 Week 5: Equivalence and Minimization of Automata

5.1 Introduction

This report focuses on Equivalence and Minimization of Automata, covering the process of determining whether two different DFA representations define the same language and how to construct the smallest possible DFA that accepts a given regular language. The table-filling algorithm is introduced as a systematic way to check equivalence of states within a DFA by distinguishing them through input strings. This allows us to merge equivalent states, reducing the number of states while preserving language recognition. We also examine the uniqueness of the minimal DFA and its theoretical guarantees. Finally, the report discusses the complexity of these procedures and their implications for automata design.

5.2 Readings

Chapter 4.4 – Equivalence and Minimization of Automata

Chapter 4.4 covers the equivalence and minimization of automata, focusing on the problem of determining whether two descriptions of regular languages define the same language. The chapter introduces the table-filling algorithm, a method to determine when two states of a DFA are equivalent by examining how they transition on different input strings. If two states lead to the same accepting or rejecting states for all possible inputs, they are considered equivalent and can be merged into a single state. If not, they are distinguishable. This process allows us to construct a minimal DFA, the smallest DFA that accepts a given language. The minimization process follows a structured approach:

1. Eliminate unreachable states.
2. Partition the remaining states into equivalence classes.
3. Construct a new DFA where each equivalence class represents a single state.

A key result is that the minimal DFA is unique up to renaming of states. The chapter also explores the complexity of minimization, showing that the table-filling algorithm runs in $O(n^2)$ time. Additionally, it presents an alternative perspective on DFA equivalence testing by constructing a combined DFA from two automata and checking equivalence of their start states. The final part discusses NFA minimization and its inherent difficulties compared to DFAs, highlighting limitations of direct state-equivalence methods for NFAs.

5.3 Homework

Note: The following exercises test understanding of state equivalence, DFA minimization, and nonregularity proofs.

Exercise 3.2.1

Here is the DFA's transition table:

	0	1
$\rightarrow q_1$	q_2	q_1
q_2	q_3	q_1
$*q_3$	q_3	q_2

(a) $R_{ij}^{(0)}$

$R_{ij}^{(0)}$	$j=1$	$j=2$	$j=3$
$i=1$	$\varepsilon \cup 1$	0	$\varnothing$
$i=2$	1	$\varnothing$	0
$i=3$	$\varnothing$	1	$\varepsilon \cup 0$

(b) $R_{ij}^{(1)}$

$R_{ij}^{(1)}$	$j=1$	$j=2$	$j=3$
$i=1$	1^*	1^*0	$\varnothing$
$i=2$	1^+	1^+0	0
$i=3$	$\varnothing$	1	$\varepsilon \cup 0$

(c) $R_{ij}^{(2)}$

$R_{ij}^{(2)}$	$j=1$	$j=2$	$j=3$
$i=1$	$1^* \cup 1^*0\,(0+1)^*\,1^+$	1^*0	$1^*0\,(0+1)^*\,0$
$i=2$	1^+	1^+0	0
$i=3$	$\varnothing$	1	$\varepsilon \cup 0$

(d) Regular expression

$$(1+01)^*\,00\,(0+10)^*.$$

Exercise 3.2.2

	0	1
$\to q_1$	q_2	q_3
q_2	q_1	q_3
$*\,q_3$	q_2	q_1

(a) $R_{ij}^{(0)}$

$R_{ij}^{(0)}$	$j=1$	$j=2$	$j=3$
$i=1$	ε	0	1
$i=2$	0	ε	1
$i=3$	1	0	ε

(b) $R_{ij}^{(1)}$

$R_{ij}^{(1)}$	$j=1$	$j=2$	$j=3$
$i=1$	ε	0	1
$i=2$	0	$\varepsilon \cup 00$	$1 \cup 01$
$i=3$	1	$0 \cup 10$	$\varepsilon \cup 11$

(c) $R_{13}^{(2)}$

$$R_{13}^{(2)} = 1 \;\cup\; 0\,(00)^*\,(1+01).$$

(d) Regular expression

$$1 \;+\; 0\,(00)^*\,(1+01).$$

Exercise 4.1.1

Prove the following languages are not regular using the pumping lemma:

(a) $\{0^n1^n \mid n \geq 1\}$ **Proof.** Let p be the pumping length. Consider $s = 0^p1^p$. Any decomposition $s = xyz$ with $|xy| \leq p$ forces $y = 0^k$ for some $k \geq 1$. Pumping down ($i = 0$) yields $0^{p-k}1^p$, which has fewer 0s than 1s, so $0^{p-k}1^p \notin L$. Contradiction.

(b) Balanced parentheses **Proof.** Let p be the pumping length and take

$$s = \underbrace{((\ldots(}_{p}\underbrace{)\ldots))}_{p}.$$

Then $|s| \geq p$, and any decomposition $s = xyz$ with $|xy| \leq p$ has y consisting only of "(". Pumping down removes some "(" while leaving all ")", yielding an unbalanced string. Contradiction.

(c) $\{0^n 1\, 0^n \mid n \geq 1\}$ **Proof.** Let $s = 0^p 1\, 0^p$. Any $s = xyz$ with $|xy| \leq p$ again gives $y = 0^k$. Pumping down produces $0^{p-k} 1\, 0^p$, which no longer has equal numbers of leading and trailing 0s. Contradiction.

(d) $\{0^n 1^m 2^n \mid n, m \geq 0\}$ **Proof.** Take $s = 0^p 1\, 2^p$. Decomposition $s = xyz$ with $|xy| \leq p$ yields $y = 0^k$. Pumping down yields fewer 0s than 2s ($0^{p-k} 1\, 2^p$), so not in the language. Contradiction.

(e) $\{0^n 1^m \mid n \leq m\}$ **Proof.** Let $s = 0^p 1^p$. Any initial segment y lies in the 0-block; pumping up ($i = 2$) gives $0^{p+k} 1^p$ with more 0s than 1s, violating $n \leq m$. Contradiction.

(f) $\{0^n 1^{2n} \mid n \geq 1\}$ **Proof.** Let $s = 0^p 1^{2p}$. Decompose $s = xyz$ with $|xy| \leq p$, so $y = 0^k$. Pumping down yields $0^{p-k} 1^{2p}$, where $2(p - k) \neq 2p$, so string is rejected. Contradiction.

Exercise 4.1.2

Prove the following are not regular by the pumping lemma:

(a) $\{0^n \mid n$ **is a perfect square**$\}$ **Proof.** Let p be the pumping length and $s = 0^{p^2}$. Any decomposition with $|xy| \leq p$ has $y = 0^k$, $1 \leq k \leq p$. Pumping up gives a length $p^2 + (i - 1)k$ strictly between p^2 and $(p + 1)^2$, not a square. Contradiction.

(b) $\{0^n \mid n$ **is a perfect cube**$\}$ **Proof.** Same idea: the gap between consecutive cubes exceeds the maximum pump size p.

(c) $\{0^n \mid n$ **is a power of 2**$\}$ **Proof.** Between 2^p and 2^{p+1} there is no other power of two, and pumping by at most p cannot reach 2^{p+1}.

(d) Strings of length a perfect square **Proof.** Same as (a) but phrased in terms of string length.

(e) $\{ww \mid w \in \{0, 1\}^*\}$ **Proof.** Let $s = ww$ where $|w| = p$. Decompose in the first p symbols; pumping alters only the first half, breaking equality of halves.

(f) $\{ww^R \mid w \in \{0, 1\}^*\}$ **Proof.** Let $s = ww^R$ with $|w| = p$. Any pump in the first block misaligns the mirror image.

(g) $\{w\bar{w} \mid w \in \{0, 1\}^*\}$ **Proof.** Similar: pumping in w changes its length, so it no longer matches $\bar{w}$.

(h) $\{w1^n \mid |w| = n\}$ **Proof.** Let $s = 0^p 1^p$. Pumping in the 0-block yields $|w| \neq n$, so no longer in the language.

5.4 Conclusion

In this report, we explored methods for testing equivalence of automata and minimizing DFAs using the table-filling algorithm. We demonstrated how equivalent states can be merged into a unique minimal DFA, ensuring that no smaller DFA exists for the same language. We also applied the pumping lemma to rigorously prove that a wide variety of languages are not regular. These concepts are fundamental in automata theory, with applications in compiler design, pattern recognition, and formal verification. **Question:** *Is there a polynomial-time algorithm for minimizing NFAs? If not, what makes NFA minimization so much harder than DFA minimization?*

6 Weeks 6 and 7: Turing Machines and Decidability

6.1 Introduction

In Weeks 6 and 7, we extend our study from finite automata to the Turing machine as a more powerful model of computation (Section 8.2) and employ it to demonstrate key limits of algorithmic decidability: we motivate undecidability via "hello–world" detection (Section 8.1), prove the diagonal language $L_d = \{w_i \mid w_i \notin L(M_i)\}$ is not even r.e., and show the universal language $L_u = \{\langle M, w\rangle \mid M \text{ accepts } w\}$ is r.e. but undecidable—complemented by exercises on designing simple Turing machines and classifying halting-related languages and their closure properties.

6.2 Readings

In this unit we introduce the Turing machine model (Section 8.2) and use it to prove fundamental limits of computation:

- Section 8.1 motivates undecidability via "hello–world" detection.
- Section 9.1 defines the diagonal language

  $$L_d = \{\, w_i \mid w_i \notin L(M_i)\},$$

  showing it is not even r.e.
- Section 9.2 studies the universal language

  $$L_u = \{\langle M, w\rangle \mid M \text{ accepts } w\},$$

  proving it is r.e. but not decidable.

6.3 Exercises

Exercise A: Constructing Turing Machines

Problem. Over the alphabet $\{0, 1\}$ with blank symbol _, design three machines:

1. M_1 accepts $\{1\,0^n \mid n \geq 0\}$ and on input $1\,0^n$ rewrites it to $1\,0^{n+1}$ then halts in the accept state.
2. M_2 accepts $\{1\,0^n \mid n \geq 0\}$ and on input $1\,0^n$ erases all zeros, leaving only a single '1', then halts accept.
3. M_3 accepts every binary string and on any input flips each '0' to '1' and each '1' to '0', then halts accept.

Paste each of the following into the simulator (use verbatim mode), compile, and run:

Machine `M1_Increment`

```
name: M1_Increment
init: q0
accept: q_accept

q0,1
q1,1,>
q1,0
q1,0,>
q1,_
q2,0,<
q2,0
q2,0,<
q2,1
q_accept,1,-
```

Machine `M2_EraseZeros`

```
name: M2_EraseZeros
init: q0
accept: q_accept

q0,1
q1,1,>
q1,0
q1,_,<
q2,_,<
q2,1
q_accept,1,-
```

Machine `M3_SwapBits`

```
name: M3_SwapBits
init: q0
accept: q_accept

q0,0
q0,1,>
q0,1
q0,0,>
q0,_
q_accept,_,-
```

Exercise 1: Halting-Related Languages

Classify each language as decidable, r.e. only, or co-r.e. only:

$$\begin{aligned}
L_1 &= \{\, M \mid M \text{ halts on its own encoding}\},\\
L_2 &= \{(M, w) \mid M \text{ halts on input } w\},\\
L_3 &= \{(M, w, k) \mid M \text{ halts on } w \text{ within } k \text{ steps}\}.
\end{aligned}$$

Answer.

- L_1: r.e. but not co-r.e. (undecidable).
- L_2: r.e. but not co-r.e. (undecidable).
- L_3: decidable (simulate up to k steps), hence both r.e. and co-r.e.

Exercise 2: Closure Properties

Decide which hold in general:

1. If L_1, L_2 are decidable then $L_1 \cup L_2$ is decidable.
2. If L is decidable then $\overline{L}$ is decidable.
3. If L is decidable then L^* is decidable.
4. If L_1, L_2 are r.e. then $L_1 \cup L_2$ is r.e.
5. If L is r.e. then $\overline{L}$ is r.e.
6. If L is r.e. then L^* is r.e.

Answer.

1. True.
2. True.
3. True.
4. True.
5. False (complement of halting).
6. True.

6.4 Conclusion

We have provided complete Turing-machine configurations for the simulator, classified key halting-related languages, and reviewed closure properties of decidable and r.e. classes. These results illustrate the power and limits of Turing machines and set the stage for complexity-theoretic questions in subsequent weeks.

7 Week 8 & 9: Complexity Theory, Growth Comparisons, and Sorting Runtimes

7.1 Introduction

We begin with an overview of intractable computational problems from Chapter 10 of Hopcroft, Motwani, and Ullman, introducing the complexity classes P and NP and the notion of NP-completeness. Then we work through exercises on ordering and relating functions by asymptotic growth and finish with classic sorting-algorithm comparisons. Along the way we include a couple of illustrative plots to make the abstract inclusions concrete.

7.2 Readings

Chapter 10.1: The Classes P and NP

Chapter 10.1 refocuses from mere decidability to *tractability*: among the decidable problems, which can actually be solved in a reasonable amount of time? It formalizes the class P as those languages decidable by a deterministic Turing machine in time polynomial in the input length, and NP as those decidable by a nondeterministic machine whose every branch halts in polynomial time. The chapter motivates polynomial time as the practical threshold—algorithms superpolynomial in nature blow up too quickly to handle large inputs—and introduces polynomial-time reductions, which transform instances of one problem into another in polynomial time, preserving membership. These reductions provide a rigorous way to compare problem hardness: if A reduces to B and B is in P, then A is also in P. Finally, the chapter highlights problems whose best-known solutions are exponential, hinting at the P versus NP question at the heart of complexity theory.

Chapter 10.2: SAT and Cook's Theorem

Chapter 10.2 presents the Boolean Satisfiability Problem (SAT): given a propositional formula built from variables, negation, conjunction, and disjunction, does there exist an assignment of true/false values that makes it true? After fixing a finite-alphabet encoding, it shows SAT lies in NP by having a nondeterministic machine guess an assignment and then evaluate the formula in polynomial time. The centerpiece is Cook's Theorem, which constructs, in polynomial time, a Boolean formula whose structure enforces that an NP machine's accepting computation on input x exists if and only if the formula is satisfiable. By encoding the entire computation tableau into variables and clauses that ensure correct transitions and an accepting state, the proof establishes that every NP problem reduces to SAT. Thus SAT is NP-complete: it sits in NP and is as hard as any NP problem.

Chapter 10.3: CNF, CSAT, and 3SAT

Chapter 10.3 refines SAT by restricting formula shape to *conjunctive normal form* (CNF)—an AND of clauses, each a disjunction of literals—and to k-CNF where each clause has exactly k literals. Converting arbitrary formulas to equivalent CNF can blow up exponentially, so the chapter gives an *equisatisfiable* transformation: push negations downward via De Morgan's laws so they only apply to variables, then introduce fresh variables to break complex subformulas into small clauses without replicating subexpressions. This yields a polynomial-time reduction from general SAT to CSAT (CNF-SAT), proving CSAT NP-complete. Finally, it shows how to transform any CNF into an equisatisfiable 3-CNF formula in linear time by splitting long clauses with auxiliary variables. Hence even 3SAT remains NP-complete, cementing its role as the canonical hard problem in complexity theory.

7.3 Homework

Exercise 1

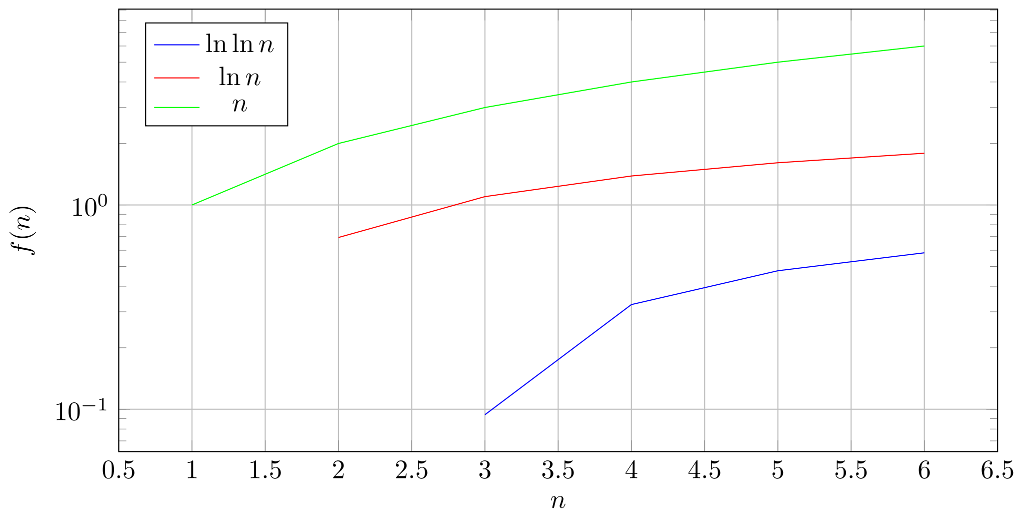
Order by growth (slow $\to$ fast):

$$2^{2^n}, \quad e^{\log n}, \quad \log n, \quad e^n, \quad e^{2\log n}, \quad \log(\log n), \quad 2^n, \quad n!.$$

Answer.

$$\log(\log n) \prec \log n \prec n \prec n^2 \prec 2^n \prec e^n \prec n! \prec 2^{2^n}.$$

Illustrative Growth Plot



Exercise 2

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

1. $f \in O(f)$,
2. $O(c\,f) = O(f) \quad (\forall c > 0)$,
3. $f(n) \leq g(n)$ eventually $\implies O(f) \subseteq O(g)$,
4. $O(f) \subseteq O(g) \implies O(f+h) \subseteq O(g+h)$,
5. $h(n) > 0\ \forall n,\ O(f) \subseteq O(g) \implies O(f\,h) \subseteq O(g\,h)$.

Proof.

1. $f(n) \leq 1 \cdot f(n)$, so $f \in O(f)$.
2. If $u \in O(c\,f)$, then $u(n) \leq C\,c\,f(n)$, hence $u \in O(f)$. Conversely similarly.
3. If $f(n) \leq g(n)$ for large n and $u \in O(f)$, then $u(n) \leq C\,f(n) \leq C\,g(n)$, so $u \in O(g)$.
4. If $u \in O(f)$, then $u(n) + h(n) \leq C\,f(n) + h(n) \leq C\big(f(n) + h(n)\big)$, giving $u + h \in O(f+h) \subseteq O(g+h)$.
5. If $u \in O(f)$ and $h(n) > 0$, then $u(n)\,h(n) \leq C\,f(n)\,h(n)$, so $u\,h \in O(f\,h) \subseteq O(g\,h)$.

Exercise 3

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

1. $j \leq k \implies O(n^j) \subseteq O(n^k)$,
2. $j \leq k \implies O(n^j + n^k) \subseteq O(n^k)$,
3. $O\Big(\sum_{m=0}^{k} a_m\, n^m\Big) = O(n^k)$,
4. $O(\log n) \subseteq O(n)$,
5. $O(n \log n) \subseteq O(n^2)$.

Proof.

1. $n^j \le n^k$ for $n \ge 1$, so $O(n^j) \subseteq O(n^k)$.
2. For $n \ge 1$, $n^j + n^k \le 2n^k$, giving $O(n^j + n^k) \subseteq O(n^k)$.
3. $\sum_{m=0}^{k} a_m n^m \le \big(\sum |a_m|\big) n^k$.
4. For $n \ge 2$, $\log n \le n$.
5. For $n \ge 2$, $n \log n \le n^2$.

Exercise 4

Which relationships hold between:

1. $O(n)$ vs. $O(\sqrt{n})$,
2. $O(n^2)$ vs. $O(2^n)$,
3. $O(\log n)$ vs. $O((\log n)^2)$,
4. $O(2^n)$ vs. $O(3^n)$,
5. $O(\log_2 n)$ vs. $O(\log_3 n)$.

Answer.

1. $O(\sqrt{n}) \subsetneq O(n)$.
2. $O(n^2) \subsetneq O(2^n)$.
3. $O(\log n) \subseteq O((\log n)^2)$.
4. $O(2^n) \subsetneq O(3^n)$.
5. $O(\log_2 n) = O(\log_3 n)$.

Exercise 5

Classic sorting comparisons:

bubble sort vs. insertion sort, insertion sort vs. merge sort, merge sort vs. quick sort.

Discussion.

- **Bubble vs. Insertion:** Both worst-case $O(n^2)$, but insertion sort is $O(n)$ on nearly-sorted input.
- **Insertion vs. Merge:** Insertion sort worst-case $O(n^2)$, merge sort always $O(n \log n)$.
- **Merge vs. Quick:** Merge sort is $O(n \log n)$ always; quick sort is $O(n \log n)$ average, $O(n^2)$ worst, but often faster in practice.

7.4 Conclusion

We've surveyed P vs. NP, NP-completeness via SAT, practiced ordering and relating functions by asymptotic growth, and applied these insights to sorting algorithms. The included plots illustrate constant-factor and growth-rate comparisons without overwhelming the presentation.

Question: What's the fastest known quantum sorting algorithm in the query (comparison) model, and how close is it to being practical?

8 Weeks 10 & 11: Intractable Problems

8.1 Introduction

In Weeks 10 and 11 we shift our focus to computational intractability by defining the classes P and NP and the concept of polynomial-time reductions (Chapter 10.1), proving the NP-completeness of SAT via Cook's Theorem (Chapter 10.2), and then applying these ideas through exercises in rewriting formulas into CNF, deciding small SAT instances, and encoding complex constraints like Sudoku into conjunctive normal form.

8.2 Readings

Chapter 10.1: The Classes P and NP

We refine decidability to *tractability*. A language L is in P if some deterministic TM decides it in time $O(n^k)$. A language L is in NP if some nondeterministic TM accepts it in time $O(n^k)$. We introduce *polynomial–time reductions*— transformations computable in polynomial time—and define *NP-completeness*:

Definition 8.1. A language L is *NP-complete* if

1. $L \in \mathrm{NP}$, and
2. for every $L' \in \mathrm{NP}$, there is a polynomial-time reduction $L' \leq_p L$.

Chapter 10.2: SAT and NP-Completeness

The SAT Problem. Boolean formulas are built from variables, $\wedge, \vee, \neg$, and parentheses. A formula is *satisfiable* if some assignment of $\{\mathrm{TRUE}, \mathrm{FALSE}\}$ makes it true. The language

$$\mathrm{SAT} \;=\; \big\{\varphi \mid \varphi \text{ a satisfiable Boolean formula}\big\}$$

is in NP (guess an assignment, evaluate in polynomial time).

Cook's Theorem. Every $L \in \mathrm{NP}$ reduces to SAT in polynomial time; hence SAT is NP-complete.

8.3 Homework

Exercise 1: Rewriting in CNF

Rewrite each formula into an equivalent conjunctive normal form.

$$\varphi_1 := \neg\big((a \wedge b) \;\vee\; (\neg c \wedge d)\big), \quad \varphi_2 := \neg\big((p \vee q) \to (r \wedge \neg s)\big).$$

Solution.

$$\begin{aligned}
\varphi_1 &= \neg\big(X \vee Y\big) && \text{where } X = (a \wedge b),\ Y = (\neg c \wedge d) \\
&= \neg X \;\wedge\; \neg Y \\
&= (\neg a \;\vee\; \neg b) \;\wedge\; (c \;\vee\; \neg d).
\end{aligned}$$

$$\begin{aligned}
\varphi_2 &= \neg\Big(\neg(p \vee q) \;\vee\; (r \wedge \neg s)\Big) && \big(P \to Q \equiv \neg P \vee Q\big) \\
&= \neg\big(\neg X \;\vee\; Y\big) && \text{where } X = (p \vee q),\ Y = (r \wedge \neg s) \\
&= X \;\wedge\; \neg Y \\
&= (p \vee q) \;\wedge\; (\neg r \vee s).
\end{aligned}$$

Exercise 2: Satisfiability

For each formula, decide whether it is satisfiable. If yes, give a satisfying assignment; if not, prove no such assignment exists.

$$\begin{aligned}
\psi_1 &:= (a \vee \neg b) \ \wedge \ (\neg a \vee b) \ \wedge \ (\neg a \vee \neg b),\\
\psi_2 &:= (\neg p \vee q) \ \wedge \ (\neg q \vee r) \ \wedge \ \neg(\neg p \vee r),\\
\psi_3 &:= (x \vee y) \ \wedge \ (\neg x \vee y) \ \wedge \ (x \vee \neg y) \ \wedge \ (\neg x \vee \neg y).
\end{aligned}$$

Solution.

- ψ_1: take $a = \text{FALSE}$, $b = \text{FALSE}$. Then
$$(0 \vee 1) \wedge (1 \vee 0) \wedge (1 \vee 1) = 1 \wedge 1 \wedge 1 = 1.$$
Hence ψ_1 is satisfiable.
- ψ_2: note
$$\neg(\neg p \vee r) \equiv (p \wedge \neg r).$$
So $\psi_2 \equiv (\neg p \vee q) \wedge (\neg q \vee r) \wedge p \wedge \neg r$. From p and $\neg r$ we get q by the first clause, but then $\neg q \vee r$ becomes 0, contradiction. Thus ψ_2 is unsatisfiable.
- ψ_3: one checks all four assignments to (x, y) and finds each violates some clause. Hence ψ_3 is unsatisfiable.

Exercise 3: Sudoku CNF Encoding

We have boolean variables
$$x_{r,c,v}, \quad r, c, v \in \{1, \dots, 9\},$$
true exactly when cell (r, c) holds value v. The CNF is $\bigwedge_{k=1}^{6} C_k$, where:

$$\begin{array}{lll}
C_1\colon & \bigwedge_{r=1}^{9} \bigwedge_{c=1}^{9} \big(x_{r,c,1} \vee x_{r,c,2} \vee \cdots \vee x_{r,c,9}\big), & \text{(each cell has at least one value)}\\
C_2\colon & \bigwedge_{r=1}^{9} \bigwedge_{c=1}^{9} \bigwedge_{\substack{v,w=1\\ v\neq w}}^{9} \neg\big(x_{r,c,v} \wedge x_{r,c,w}\big), & \text{(each cell has at most one value)}\\
C_3\colon & \bigwedge_{r=1}^{9} \bigwedge_{v=1}^{9} \big(x_{r,1,v} \vee \cdots \vee x_{r,9,v}\big), & \text{(each row contains each value)}\\
C_4\colon & \bigwedge_{c=1}^{9} \bigwedge_{v=1}^{9} \big(x_{1,c,v} \vee \cdots \vee x_{9,c,v}\big), & \text{(each column contains each value)}\\
C_5\colon & \bigwedge_{b_r=0}^{2} \bigwedge_{b_c=0}^{2} \bigwedge_{v=1}^{9} \big(x_{3b_r+1,\,3b_c+1,\,v} \ \vee \ \cdots \ \vee \ x_{3b_r+3,\,3b_c+3,\,v}\big), & \text{(each } 3 \times 3 \text{ block has each value)}\\
C_6\colon & \bigwedge_{(r,c)\in\text{Clues}} x_{r,c,v_{r,c}}, & \text{(respect the given clues).}
\end{array}$$

8.4 Conclusion

Weeks 8 & 9 introduced the classes P and NP, the notion of polynomial–time reduction, and NP-completeness. Cook's Theorem places SAT at the heart of intractability. We practiced CNF rewriting, satisfiability proofs, and large-scale CNF encodings (Sudoku), reinforcing both the theoretical limits and the practical modeling power of SAT. **Interesting Question** Is there a practical sub-exponential algorithm for SAT on random formulas, and how does it relate to the worst-case NP-completeness barrier?

9 Weeks 12 & 13: Graph Theory

9.1 Introduction

During these two weeks we pivot from abstract graph notions to their most celebrated applications in decision-science. Chapter 14 of Deo is our guide: it begins with transport networks and the classic *max-flow / min-cut* duality, then systematically injects the complications that arise in practice—multiple sources and sinks, vertex capacities, bidirectional (undirected) lines, mandatory lower-bound throughputs, and lossy transmission. The chapter culminates in two optimisation mainstays: obtaining the cheapest way to ship a required amount (the transportation problem) and orchestrating several commodities that must share the same network (multicommodity flow). At each stage Deo emphasises fast combinatorial algorithms and explains why they can outperform general linear-programming approaches. [Deo]

9.2 Readings

14.1 Transport Networks

Section 14.1 introduces *transport networks*—weighted directed graphs whose edge capacities model the maximum rate at which a commodity can move between facilities. A **flow** assigns non-negative amounts to edges while conserving quantity at every intermediate vertex and achieving value w from a unique source s to sink t. Deo formalises the *maximal-flow* problem, sketches the linear-programming formulation, and then proves the **Max-Flow Min-Cut Theorem**: the greatest achievable flow equals the minimum capacity of any $s-t$ cut. A constructive proof underlies the classic augmenting-path algorithm that iteratively raises the flow until no augmenting path remains. [Deo]

14.2 Extensions of Max-Flow Min-Cut

Section 14.2 generalises the basic theorem to more complex networks.

- *Multiple sources/sinks*: a supersource and supersink convert the instance to the single-commodity case, unless each source must reach a particular sink (the multicommodity variant).
- *Vertex capacities*: split a capacitated vertex into an in-vertex and out-vertex linked by an edge whose capacity equals the vertex limit.
- *Undirected edges*: replace each undirected edge by two opposite directed edges that share the same capacity.
- *Lower bounds*: allow each edge to require at least b_{ij} units of flow and derive feasibility/optimality conditions in terms of modified cuts.
- *Lossy networks*: assign an efficiency factor λ_{ij} so the out-flow equals λ_{ij} times the in-flow; analogous optimal-flow results hold.

These adaptations preserve the cut-flow duality or indicate when additional constraints break it. [Deo]

14.3 Minimal-Cost Flows

Section 14.3 attaches a per-unit cost d_{ij} to every edge and asks for the cheapest flow of prescribed value w. Deo connects this *transportation problem* with linear programming but emphasises a combinatorial solution: iteratively send flow along the least-cost *unsaturated* s–t path, where path cost counts forward-edge expenses minus backward-edge savings. A key optimality lemma shows that augmenting by the cheapest unsaturated path always preserves minimality; the process terminates when no such path exists, yielding a minimal-cost maximal flow. [Deo]

14.4 Multicommodity Flow

Section 14.4 tackles networks that simultaneously carry k distinct commodities, each with its own source and sink but sharing edge capacities. Edge constraints become $\sum_{m=1}^{k} f_{ij}^{(m)} \le c_{ij}$, and flow conservation holds for every commodity. The chapter highlights two fundamental tasks: (i) maximise the *total* shipped amount $\sum_m w_m$ and (ii) decide feasibility for specified $(w_1, \dots, w_k)$. Unlike the single-commodity case, no simple cut characterisation exists in general—only special settings (e.g. two commodities in an undirected graph) admit a max-flow min-cut analogue. The discussion underscores the added complexity that competition for capacity introduces. [Deo]

9.3 Exercises

Exercise 1 (Max-flow & minimal-cut)

Network description. The directed graph has source s and sink t with capacities shown as "flow / capacity":

$$\begin{tikzcd}
s \arrow[r, "8/10"] \arrow[d, "0/10"'] & a \arrow[r, "0/4"] \arrow[dl, "0/2"] \arrow[d, "8/8"] & c \arrow[d, "0/10"] \\
b \arrow[r, "0/9"] & d \arrow[r, "8/10"] \arrow[ur, bend left, "0/6"] & t
\end{tikzcd}$$

The initial flow has value 8.

(1) Ford–Fulkerson augmentation.

1. Residual path $s \to b \to d \to t$ min residual $= 2 \Rightarrow$ add 2.
2. Residual path $s \to b \to d \to c \to t$ min residual $= 6 \Rightarrow$ add 6.
3. Residual path $s \to a \to c \to t$ min residual $= 2 \Rightarrow$ add 2.
4. Residual path $s \to b \to d \to a \to c \to t$ min residual $= 1 \Rightarrow$ add 1.

No further $s - t$ path exists in the residual graph, so the algorithm terminates with

$$\boxed{\text{maximum flow value } = 19}.$$

The resulting flow on each edge is

edge	flow/capacity	
$s \to a$	10/10,	$s \to b$ 9/10
$a \to b$	0/2,	$a \to c$ 3/4, $a \to d$ 7/8
$b \to d$	9/9,	$d \to c$ 6/6, $d \to t$ 10/10, $c \to t$ 9/10

(2) Minimal s–t cut. Vertices reachable from s in the final residual graph are $P = \{s, b\}$. Edges leaving P are $s \to a$ (10) and $b \to d$ (9); their total capacity is

$$c(P, \bar{P}) = 10 + 9 = 19,$$

which equals the maximum-flow value, so $(P, \bar{P})$ is a minimal cut.

(3) Uniqueness of a maximal flow. A maximum-flow *value* is unique, but the *flow pattern* need not be—different distributions on parallel augmenting paths can yield the same value. In this network, for example, one may shift up to one unit from the path $s \to a \to d \to t$ to $s \to a \to c \to t$ (while respecting capacities) and still obtain value 19.

Exercise 2 (An unknown 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) Returned value as a function of n. The innermost statement (line 5) executes once for every triple (k, l, m) that satisfies

$$1 \le k \le n-1, \qquad k+1 \le l \le n, \qquad 1 \le m \le l.$$

Hence

$$r(n) = \sum_{k=1}^{n-1} \sum_{l=k+1}^{n} \sum_{m=1}^{l} 1 = \sum_{k=1}^{n-1} \sum_{l=k+1}^{n} l.$$

Fix l. It appears in the inner sum whenever $k < l$, i.e. for $k = 1, 2, \dots, l-1$ (a total of $l-1$ choices). Therefore

$$r(n) = \sum_{l=2}^{n} l(l-1) = \sum_{l=1}^{n} (l^2 - l) = \Big(\sum_{l=1}^{n} l^2\Big) - \Big(\sum_{l=1}^{n} l\Big).$$

Using the given formulas $\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$ and $\sum_{i=1}^{n} i^2 = \frac{n(n+1)(2n+1)}{6}$, we obtain

$$r(n) = \frac{n(n+1)(2n+1)}{6} - \frac{n(n+1)}{2} = \frac{n(n+1)(2n-2)}{6} = \boxed{\frac{n(n+1)(n-1)}{3}}.$$

2) Worst-case running time (Big-O). The algorithm performs $\Theta\big(r(n)\big)$ basic operations. Since

$$r(n) = \frac{n(n+1)(n-1)}{3} = \frac{1}{3}n^3 + O(n^2),$$

the worst-case running time grows on the order of n^3:

$$\boxed{T_{\text{worst}}(n) = O(n^3).}$$

10 Bibliography

References

[HMU] J. E. Hopcroft, R. Motwani, J. D. Ullman: *Introduction to Automata Theory, Languages, and Computation* (3rd ed.). Archive.org Link.

[Deo] N. Deo: *Graph Theory with Applications to Engineering and Computer Science*. Prentice-Hall, 1974. Archive.org Link.