CPSC-406 Course Report

Traehan Arnold
Chapman University

May 25, 2025

Abstract

This document serves as a cumulative report for the CPSC-406 course. Each week includes homework solutions, deeper explorations of the topics, and original questions aimed at reinforcing understanding and inspiring further inquiry. The structure follows course guidelines and demonstrates proficiency in technical writing and mathematical reasoning using LaTeX.

Contents

1 Introduction

This report is a collection of what I've learned and worked on throughout the CPSC-406 course. Each week, I completed assignments that helped me understand new ideas in computer science—from how machines read and process input, to how we measure the speed and limits of algorithms. Along the way, I included my own thoughts and questions to explore the material more deeply.

We started with basic ideas like finite automata and regular expressions, and gradually moved into more advanced topics like Turing machines, undecidability, and algorithm analysis. Some problems focused on theory, like proving things with induction or breaking down how machines work. Others were more hands-on, like using Ford-Fulkerson to find the max flow in a network or turning logic puzzles like Sudoku into code-like rules.

I also wrote a reflection called *"Beyond the Code: Understanding the Essence of Algorithms"* where I share how this class helped me see that writing good code isn't just about syntax—it's about solving problems clearly, efficiently, and thoughtfully.

Overall, this report shows how much I've grown in understanding what computer science is really about: not just programming, but thinking deeply about how problems are solved—and how we can build tools to do that better.

2 Week 1: Intro To Automata

2.1 Homework

- **Problem 1:** Vending Machine Automaton
  Determine the accepted words for reaching state 25

```
accepted_words_vending_machine = [
    "55555",
    "55510",
    "55105",
    "51055",
    "10555",
    "10105"
]
```

  Any sequence of 5s and 10s summing to 25 is accepted.

- **Problem 2:** Turnstile Automaton
  Describe accepted words via a regular expression

```
regular_expression_turnstile = r"pay (pay | push pay)*"
```

  Explanation:

  - `pay` must be the first input to unlock the turnstile.
  - Additional `pay` keeps it unlocked.
  - `push pay` locks and then unlocks, repeating indefinitely.

- **Problem 3:** Binary Language Classification
  Determine which words belong to L1, L2, and L3.

```
binary_language_classification = {
"10011": {"L1": False, "L2": False, "L3": False},
"100": {"L1": False, "L2": False, "L3": False},
"10100100": {"L1": True, "L2": True, "L3": True},
"1010011100": {"L1": True, "L2": False, "L3": True},
"11110000": {"L1": False, "L2": True, "L3": True}
}
```

- **Problem 4:** DFA Accepting States
  Determine which words end in the accepting state q1.

```
    dfa_accepting_states = {
    "0010": True,
    "1101": True,
    "1100": False
}
```

Words `"0010"` and `"1101"` end in q1, while `"1100"` does not.

2.2 Exploration

The introduction to automata theory lays the foundation for understanding how abstract machines can be used to model computation. At its core, automata theory provides a framework for recognizing patterns and processing input based on defined rules. Deterministic Finite Automata (DFAs) are machines where each state has exactly one transition for every input symbol, making them predictable and easy to implement in practice—for example, in lexical analyzers within compilers or in validating input formats like dates or phone numbers. Non-deterministic Finite Automata (NFAs), on the other hand, allow multiple or zero transitions for the same symbol from a given state, representing a more flexible and intuitive model of choice and ambiguity in computation. While NFAs are theoretically equivalent in power to DFAs, they often lead to more concise designs and are foundational in the implementation of tools like regular expression engines. This contrast between DFAs and NFAs reveals a powerful insight: non-determinism, though conceptually more complex, can simplify design and reveal underlying symmetries in language structure, even when determinism is needed for implementation.

2.3 Question

In the beginning, how did the development of automata theory influence early computing and programming languages, and were there any key breakthroughs that directly shaped how we design and understand computation today?

3 Week 3: Operations on Automata

3.1 Homework

- **Problem 1:** Description of the Language Accepted by $\mathcal{A}^{(2)}$ The automaton $\mathcal{A}^{(2)}$ accepts the language:

$$L(\mathcal{A}^{(2)}) = \{w \in \{a, b\}^* \mid |w| \text{ is odd and every letter at an odd position is } a\}$$

That is, all words over the alphabet $\{a, b\}$ that have an odd length, and every letter in an odd-numbered position (1st, 3rd, 5th, etc.) is the letter a.

2. Computation of Extended Transition Functions

a) For $\mathcal{A}^{(1)}$, compute $\widehat{\delta^{(1)}}(1, abaa)$

Given that $\mathcal{A}^{(1)}$ accepts non-empty words where no two consecutive letters are the same, and assuming the states are numbered, we proceed step by step:

- Start at state 1
- Read a: move to state 2
- Read b: move to state 1
- Read a: move to state 2
- Read a: since the previous letter was also a, and two consecutive letters are the same, this transition is invalid.

Therefore, the computation halts, and the word $abaa$ is rejected by $\mathcal{A}^{(1)}$.

b) For $\mathcal{A}^{(2)}$, compute $\widehat{\delta^{(2)}}(1, abba)$

Assuming the transition function $\delta^{(2)}$ is defined as per the automaton's description:

- Start at state 1
- Read a: move to state 2
- Read b: move to state 3
- Read b: move to state 2
- Read a: move to state 3

After processing the entire input $abba$, the automaton ends in state 3. If state 3 is an accepting state, then the word is accepted; otherwise, it is rejected. Based on the language description, since $abba$ has even length and the language requires odd length, $abba$ is rejected by $\mathcal{A}^{(2)}$.

Problem Statement

Let n be a natural number, and let $P(m)$ be a property pertaining to the natural numbers such that whenever $P(m)$ is true, $P(m++)$ is also true. Show that if $P(n)$ is true, then $P(m)$ is true for all $m \geq n$. This principle is sometimes referred to as the principle of induction starting from the base case n.

Solution

We aim to prove that if $P(n)$ is true and $P(m) \Rightarrow P(m++)$ holds for all $m \in \mathbb{N}$, then $P(m)$ is true for all $m \geq n$.

Approach

Define a new property $Q(k) := P(n+k)$ for $k \in \mathbb{N}$. Our goal is to show that $Q(k)$ is true for all $k \in \mathbb{N}$, which would imply that $P(m)$ is true for all $m \geq n$.

Base Case

For $k = 0$, we have $Q(0) = P(n+0) = P(n)$, which is given to be true.

Inductive Step

Assume $Q(k)$ is true for some $k \in \mathbb{N}$, i.e., $P(n+k)$ is true. Since $P(m) \Rightarrow P(m++)$ for all m, it follows that $P(n+k) \Rightarrow P((n+k)++) = P(n+k+1)$. Therefore, $Q(k+1) = P(n+k+1)$ is true.

Conclusion

By the principle of mathematical induction, $Q(k)$ is true for all $k \in \mathbb{N}$. Consequently, $P(m)$ is true for all $m \geq n$.

4 Week 4: NFAs and Determinization

4.1 Homework

- **Step 1: Viewing DFA as an NFA** A DFA is a special case of an NFA where each state has exactly one transition per input symbol. Since NFAs allow multiple (or no) transitions for a symbol, any DFA can be interpreted as an NFA by wrapping each transition in a singleton set.
- **Step 2: Constructing NFA A' from DFA A**

  Given:

  - Start state: 1
  - Transitions:

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

  Then the equivalent NFA A' is defined as:

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

  with:

$$Q' = \{1,2,3\}, \quad \Sigma = \{a,b\}, \quad \delta'(q,x) = \{\delta(q,x)\}, \quad q_0' = 1, \quad F' = \{3\}$$

- **Step 3: Justification**

  Since each transition in the DFA becomes a singleton set in the NFA, A and A' recognize the same language:

$$L(A) = L(A')$$

- **Language Accepted by A**

  The NFA A accepts all binary strings that contain at least one occurrence of the substring "00" or end in state q_3.

- **Specification of A**
  - $Q = \{q_0, q_1, q_2, q_3\}$
  - $\Sigma = \{0, 1\}$
  - Start state: q_0
  - Accepting state: $F = \{q_3\}$
  - Transition function:

δ	0	1
q_0	$\{q_0\}$	$\{q_0, q_1\}$
q_1	$\emptyset$	$\{q_2\}$
q_2	$\{q_1, q_3\}$	$\emptyset$
q_3	$\{q_3\}$	$\{q_3\}$

- **Extended Transition Function $\widehat{\delta}(q_0, 10110)$**

  Step-by-step:

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

  Since q_3 is in the final set, the word `10110` is accepted.

- **Paths for $v = 1100$ and $w = 1010$**
  - $v = 1100$ reaches q_3, so it is accepted.
  - $w = 1010$ does not reach q_3, so it is rejected.

- **Determinization: DFA A_D**

  Using the power set construction, the DFA states are subsets of $\{q_0, q_1, q_2, q_3\}$. The accepting states are those that include q_3.

- **Minimization of A_D**

  Since $L(A) = L(A_D)$, we apply minimization to merge equivalent states and obtain the smallest DFA that accepts the same language.

4.2 Exploration

This assignment deepened the understanding of how NFAs and DFAs relate, particularly through the power set construction and the minimization process. Viewing DFAs as restricted NFAs reveals a hierarchy of expressive power that is theoretically equal, but practically distinct. Minimization emphasizes efficiency in state machines—a crucial aspect in compiler optimization and pattern recognition.

4.3 Question

In practice, how does determinization affect the efficiency of automata-based tools (e.g., regex engines), and are there modern alternatives to the power set construction to handle large or infinite state spaces more efficiently?

5 Week 5: Regular Expressions and DFA Minimization

5.1 Homework

- **Exercise 3.2.1: DFA Regular Expressions and State Elimination**

  Given Transition Table:

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

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

$$\begin{array}{lll} R_{11}^{(0)} = \varepsilon & R_{12}^{(0)} = 0 & R_{13}^{(0)} = \emptyset \\ R_{21}^{(0)} = 1 & R_{22}^{(0)} = \varepsilon & R_{23}^{(0)} = 0 \\ R_{31}^{(0)} = \emptyset & R_{32}^{(0)} = 1 & R_{33}^{(0)} = 0 \mid \varepsilon \end{array}$$

  b) Regular Expressions $R_{ij}^{(1)}$ (using q_1 as intermediate)

$$\begin{array}{lll} R_{11}^{(1)} = \varepsilon & R_{12}^{(1)} = 0 & R_{13}^{(1)} = \emptyset \\ R_{21}^{(1)} = 1 & R_{22}^{(1)} = \varepsilon \mid 10 & R_{23}^{(1)} = 0 \\ R_{31}^{(1)} = \emptyset & R_{32}^{(1)} = 1 & R_{33}^{(1)} = 0 \end{array}$$

  c) Regular Expressions $R_{ij}^{(2)}$ (using q_1 and q_2 as intermediate)

$$R_{11}^{(2)} = \varepsilon \mid 0(10)^*1$$
$$R_{12}^{(2)} = 0 \mid 0(10)^*10$$
$$R_{13}^{(2)} = 0(10)^*0$$
$$R_{23}^{(2)} = 0 \mid 10(10)^*0$$
$$R_{33}^{(2)} = 0 \mid 1(10)^*0$$

d) Final Regular Expression:

$$R = 0(10)^*0$$

e) State Elimination Method:

After eliminating q_2, we derive:

$$R = (1 \mid 01)00(0)$$

- **Exercise 3.2.2: Another DFA Regular Expression Conversion**

  Given Transition Table:

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

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

$$R_{11}^{(0)} = \varepsilon \qquad R_{12}^{(0)} = 0 \qquad R_{13}^{(0)} = 1$$
$$R_{21}^{(0)} = 0 \qquad R_{22}^{(0)} = \varepsilon \qquad R_{23}^{(0)} = 1$$
$$R_{31}^{(0)} = 1 \qquad R_{32}^{(0)} = 0 \qquad R_{33}^{(0)} = \varepsilon$$

b) Regular Expressions $R_{ij}^{(1)}$ (using q_1 as intermediate)

$$R_{12}^{(1)} = 0 \qquad R_{13}^{(1)} = 1$$
$$R_{22}^{(1)} = \varepsilon \mid 00 \qquad R_{23}^{(1)} = 1 \mid 01$$
$$R_{33}^{(1)} = \varepsilon \mid 11$$

c) Final Regular Expression (from q_1 to q_3):

$$R = 1 \mid 0(00)^*(1 \mid 01)$$

d) State Elimination Result:

$$R = (1 \mid 0(00)^*1)$$

- **Exercise 4.4.1: DFA Minimization**

  Given Transition Table:

	0	1
$\rightarrow A$	B	A
B	A	C
C	D	B
$*D$	D	A
E	D	F
F	G	E
G	F	G
H	G	D

  a) Distinguishability Table:

  Mark all pairs with one final and one non-final state. States E, F, G are indistinguishable and can be merged.

  b) Minimized DFA:

  Merge $E, F, G \rightarrow$ EFG

State	0	1	Accept?
$\rightarrow A$	B	A	No
B	A	C	No
C	D	B	No
D	D	A	Yes
EFG	EFG	EFG	No
H	EFG	D	No

- **Exercise 4.4.2: DFA Minimization (Larger Example)**

  Given Transition Table:

	0	1
$\to A$	B	E
B	C	F
$*C$	D	H
D	E	I
E	F	H
$*F$	G	B
G	H	B
H	I	C
$*I$	A	E

a) Distinguishability Table:

Final states are C, F, I. These are found to be equivalent and merged.

b) Minimized DFA:

Merge $C, F, I \to$ CFI

State	0	1	Accept?
$\to A$	B	E	No
B	CFI	CFI	No
CFI	D	H	Yes
D	E	CFI	No
E	CFI	H	No
G	H	B	No
H	CFI	CFI	No

5.2 Exploration

This week's homework emphasized the process of converting DFAs to regular expressions and minimizing DFAs using both the state elimination and distinguishability table methods. By systematically eliminating states or computing equivalent expressions through dynamic construction, we deepen our grasp on how regular languages can be represented and optimized both algebraically and structurally. Minimization shows the elegance and efficiency of theoretical computer science when reducing redundancy.

5.3 Question

Given the exponential blow-up risk in DFA to regular expression conversions, what are some modern tools or heuristics used in compilers or regex engines to efficiently simplify or avoid full state elimination?

6 Week 6: Turing Machines

6.1 Homework

- **Problem 1: TM for $L = \{10^n : n \in \mathbb{N}\}$ and return 10^{n+1}**

Idea: The Turing Machine (TM) scans the input 10^n, moves to the end of the string, and appends an extra 0 at the end.

- **States:** q_0 (start), q_1 (move right to the end), q_2 (write extra 0), q_{accept}
- **Alphabet:** {0, 1, B}
- **Transitions:**

$$\delta(q_0, 1) = (q_1, 1, R)$$
$$\delta(q_1, 0) = (q_1, 0, R)$$
$$\delta(q_1, B) = (q_2, 0, S)$$
$$\delta(q_2, 0) = (q_{accept}, 0, S)$$

- **Problem 2: TM for $L = \{10^n : n \in \mathbb{N}\}$ and return 1**

  Idea: The TM erases all 0s and leaves only the initial 1.

  - **States:** q_0 (start), q_1 (erase 0s), q_2 (go back left), q_3 (halt with 1), q_{accept}
  - **Alphabet:** {0, 1, B}
  - **Transitions:**

$$\delta(q_0, 1) = (q_1, 1, R)$$
$$\delta(q_1, 0) = (q_1, B, R)$$
$$\delta(q_1, B) = (q_2, B, L)$$
$$\delta(q_2, 1) = (q_3, 1, S)$$
$$\delta(q_3, 1) = (q_{accept}, 1, S)$$

- **Problem 3: TM that swaps 0s and 1s in a binary string**

  Idea: The machine first reads left to right and replaces 0 with a temporary marker X and 1 with Y. Then it returns to the beginning and replaces $X \to 1$ and $Y \to 0$.

  - **States:** q_0 (marking), q_1 (return left), q_2 (finalizing), q_{accept}
  - **Transitions:**

$$\delta(q_0, 0) = (q_0, X, R)$$
$$\delta(q_0, 1) = (q_0, Y, R)$$
$$\delta(q_0, B) = (q_1, B, L)$$
$$\delta(q_1, X) = (q_1, X, L)$$
$$\delta(q_1, Y) = (q_1, Y, L)$$
$$\delta(q_1, B) = (q_2, B, R)$$
$$\delta(q_2, X) = (q_2, 1, R)$$
$$\delta(q_2, Y) = (q_2, 0, R)$$
$$\delta(q_2, B) = (q_{accept}, B, S)$$

6.2 Exploration

This homework introduces foundational Turing Machine operations, from simple appending and deletion tasks to full string transformation. These problems help illustrate how abstract machines manipulate symbols on a tape to perform logic-driven operations. The clarity of the TM transition design makes it easier to see the progression from finite automata to Turing-complete models capable of full computation.

6.3 Question

What optimizations exist in real-world implementations of Turing-equivalent machines (e.g., CPU architectures) to avoid the inefficiencies of multi-pass operations like the two-phase symbol replacement in Problem 3?

7 Week 7: Decidability and Language Classification

7.1 Homework

- **Exercise 1: Classifying Languages**

  Each language is analyzed based on its position in the hierarchy of decidable, recursively enumerable (r.e.), and co-r.e. languages.

  1. $L_1 := \{M \mid M \text{ halts on itself}\}$
     - This is a version of the *diagonal halting problem.*
     - **Classification:**
       * Not decidable (reducible from the halting problem),
       * Not r.e. (no TM can semi-decide it),
       * Not co-r.e. (its complement is not r.e.).
  2. $L_2 := \{(M, w) \mid M \text{ halts on } w\}$
     - This is the classic *Halting Problem.*
     - **Classification:**
       * Not decidable,
       * Recursively enumerable (r.e.) — simulate M on w,
       * Not co-r.e.
  3. $L_3 := \{(M, w, k) \mid M \text{ halts on } w \text{ in at most } k \text{ steps}\}$
     - **Classification:**
       * Decidable — simulate M on w for at most k steps.

- **Exercise 2: True/False on Closure Properties**

  1. **True.** The union of two decidable languages is decidable. *(Run both deciders; accept if either accepts.)*

2. **True.** The class of decidable languages is closed under complement. *(Flip accept/reject in the decider.)*
3. **True.** If L is decidable, then L^* is decidable. *(Nondeterministically split input and test each part with L's decider.)*
4. **True.** The union of two r.e. languages is r.e. *(Use dovetailing to run both TMs.)*
5. **False.** The complement of an r.e. language is not necessarily r.e. *(Counterexample: the Halting Problem is r.e., but its complement is not.)*
6. **False.** L^* may not be r.e. even if L is. *(Let $L = \{w \mid M_w \text{ halts}\}$; then L^* is not necessarily r.e.)*

7.2 Exploration

This assignment explores the boundaries of computation by classifying problems based on their decidability and enumerability. Distinguishing between r.e. and co-r.e. languages sharpens our understanding of what Turing machines can do — both in terms of acceptance and rejection. These concepts underpin the limits of algorithmic verification, particularly in fields like software verification and logic where halting behavior cannot always be determined.

7.3 Question

How do modern programming language safety checks and static analysis tools navigate the undecidability of halting and related problems, and what compromises do they make to provide useful results in practice?

8 Weeks 8 & 9: Asymptotic Analysis and Growth Rate Comparisons

8.1 Homework

- **Exercise 1: Order of Growth Comparison**

  Order the following functions from slowest to fastest:

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

  Classification:

  - **Slowest:** $\log(\log n)$ (double logarithmic)
  - $\log n$ (logarithmic)
  - $e^{\log n} = n$
  - $e^{2\log n} = n^2$
  - 2^n
  - e^n

- $n!$ (super-exponential)
- **Fastest:** 2^{2^n} (double exponential)

- **Exercise 2: Proving Big-O Properties**

  Let $f, g, h : \mathbb{N} \to \mathbb{R}_{\geq 0}$. Prove the following:

  1. $f \in O(f)$ *(Trivially true with constant $c = 1$)*
  2. If $c > 0$, then $O(c \cdot f) = O(f)$ *(Multiplicative constants do not affect Big-O classes)*
  3. If $f(n) \leq g(n)$ for large n, then $O(f) \subseteq O(g)$
  4. If $O(f) \subseteq O(g)$, then $O(f + h) \subseteq O(g + h)$
  5. If $h(n) > 0$ and $O(f) \subseteq O(g)$, then $O(f \cdot h) \subseteq O(g \cdot h)$

- **Exercise 3: Polynomial Growth Comparisons**

  1. If $j \leq k$, then $O(n^j) \subseteq O(n^k)$
  2. If $j \leq k$, then $O(n^j + n^k) \subseteq O(n^k)$
  3. $O\left(\sum_{i=0}^{k} a_i n^i\right) = O(n^k)$ *(Highest-degree term dominates)*
  4. $O(\log n) \subseteq O(n)$
  5. $O(n \log n) \subseteq O(n^2)$

- **Exercise 4: Comparing Growth Classes**

  1. $O(n) \supset O(\sqrt{n})$ *(Square root grows slower)*
  2. $O(n^2) \subset O(2^n)$ *(Exponential outpaces polynomial)*
  3. $O(\log n) \subset O(\log^2 n)$
  4. $O(2^n) \subset O(3^n)$
  5. $O(\log^2 n) = O(\log^3 n)$ *(Difference is constant factor; change-of-base)*

- **Exercise 5: Sorting Algorithm Runtime Comparison**
  - **Bubble Sort vs Insertion Sort:** Both are $O(n^2)$, but insertion sort is generally faster in practice due to fewer swaps.
  - **Insertion Sort vs Merge Sort:** Merge sort is asymptotically faster: $O(n \log n)$ vs $O(n^2)$.
  - **Merge Sort vs Quick Sort:** Both average $O(n \log n)$, but quick sort has a worse-case of $O(n^2)$ unless properly optimized (e.g., randomized pivots or tail call optimizations).

8.2 Exploration

This dual-week homework provided a rigorous comparison of function growth classes through asymptotic analysis. It reinforces the idea that constants and lower-order terms become irrelevant as inputs scale, and shows how subtle differences in logarithmic and exponential bases impact performance. The comparative sorting algorithm analysis emphasizes the real-world importance of choosing optimal algorithms, especially when average and worst-case complexities diverge.

8.3 Question

In algorithm design, how do practitioners balance worst-case guarantees against average-case efficiency, especially when the faster algorithm has a higher risk of worst-case performance (e.g., Quick Sort vs Merge Sort)?

9 Weeks 10 & 11: CNF Conversion, Satisfiability, and Sudoku Encoding

9.1 Homework

- **Exercise 1: Convert to CNF (Conjunctive Normal Form)**

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

$$\begin{aligned}\varphi_1 &= \neg((a \wedge b) \vee (\neg c \wedge d)) \\ &= \neg(a \wedge b) \wedge \neg(\neg c \wedge d) \quad \text{(De Morgan's Law)} \\ &= (\neg a \vee \neg b) \wedge (c \vee \neg d)\end{aligned}$$

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

$$\begin{aligned}\varphi_2 &= \neg(\neg(p \vee q) \vee (r \wedge \neg s)) \\ &= \neg((\neg p \wedge \neg q) \vee (r \wedge \neg s)) \quad \text{(De Morgan)} \\ &= \neg(\neg p \wedge \neg q) \wedge \neg(r \wedge \neg s) \\ &= (p \vee q) \wedge (\neg r \vee s)\end{aligned}$$

- **Exercise 2: Propositional Satisfiability**

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

     Test assignment: $a = 0, b = 0$

$$\begin{aligned}(a \vee \neg b) &= 0 \vee 1 = 1 \\ (\neg a \vee b) &= 1 \vee 0 = 1 \\ (\neg a \vee \neg b) &= 1 \vee 1 = 1\end{aligned}$$

     Satisfiable. Valid assignment: $a = 0, b = 0$

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

     Rewrite last clause:

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

From $p \wedge \neg r$, we need $q = 1$ from $\neg p \vee q$, and $\neg q = 0 \Rightarrow r = 1$ from $\neg q \vee r$, which contradicts $\neg r$. **Unsatisfiable**.

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

   Test all four truth assignments:

   - $x = 0, y = 0 \Rightarrow (x \vee y) = 0$
   - $x = 0, y = 1 \Rightarrow (x \vee \neg y) = 0$
   - $x = 1, y = 0 \Rightarrow (\neg x \vee y) = 0$
   - $x = 1, y = 1 \Rightarrow (\neg x \vee \neg y) = 0$

   Unsatisfiable.

- **Exercise 3: Encoding Sudoku Constraints in CNF**

  Let $x_{r,c,v}$ be true if cell (r, c) contains value v, where $r, c, v \in \{1, \ldots, 9\}$. The full CNF formula is:

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

  - C_1: Each cell must contain **at least one** value

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

  - C_2: Each cell contains **at most one** value

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

  - C_3: Each **row** contains every number

$$C_3 := \bigwedge_{r=1}^{9} \bigwedge_{v=1}^{9} \left(\bigvee_{c=1}^{9} x_{r,c,v} \right)$$

  - C_4: Each **column** contains every number

$$C_4 := \bigwedge_{c=1}^{9} \bigwedge_{v=1}^{9} \left(\bigvee_{r=1}^{9} x_{r,c,v} \right)$$

  - C_5: Each 3x3 **block** contains every number

    Let block coordinates be indexed by $br, bc \in \{0, 1, 2\}$

$$C_5 := \bigwedge_{v=1}^{9} \bigwedge_{br=0}^{2} \bigwedge_{bc=0}^{2} \left(\bigvee_{r=3br+1}^{3br+3} \bigvee_{c=3bc+1}^{3bc+3} x_{r,c,v} \right)$$

– C_6: Encode the **given clues**

For known values $G = \{(r_i, c_i, v_i)\}$

$$C_6 := \bigwedge_{(r_i,c_i,v_i)\in G} x_{r_i,c_i,v_i}$$

9.2 Exploration

This homework focused on translating logical formulas into Conjunctive Normal Form (CNF) and understanding satisfiability. The systematic breakdown of Sudoku encoding into CNF also highlights how formal logic underpins constraint satisfaction problems, especially in artificial intelligence and automated reasoning. The ability to represent puzzles like Sudoku in logic illustrates the expressive power of propositional formulas.

9.3 Question

How do modern SAT solvers efficiently handle real-world CNF encodings with millions of variables, and what optimizations make them practical for applications like AI planning and verification?

10 Weeks 12 & 13: Maximal Flow and Minimal Cut

10.1 Homework

- **Problem: Ford-Fulkerson Maximum Flow Algorithm**

  Given a flow network G, compute the maximal flow using the Ford-Fulkerson method.

  1. **Augmenting Path 1:** $s \to a \to b \to d \to t$

     Bottleneck capacity:

     $$\min(8, 2, 8, 10) = 2$$

     Update flows accordingly.

  2. **Augmenting Path 2:** $s \to a \to c \to d \to t$

     Bottleneck:

     $$\min(6, 4, 6, 10) = 4$$

     Update flows.

  3. **Augmenting Path 3:** $s \to b \to d \to t$

     Bottleneck:

     $$\min(10, 6, 10) = 6$$

     Update flows.

  Total flow into t: $2 + 4 + 6 = 12$ **Maximum flow:**

  $$\boxed{12}$$

- **Minimal Cut Analysis**

  After completing Ford-Fulkerson, the reachable nodes from s in the residual graph are:

  $$\{s\}$$

  So, one minimal cut separates s from t by removing edges from $\{s\}$ to $\{a, b, c, d, t\}$:

  Cut edges: $s \to a$, $s \to b$ **Cut capacity:** $10 + 10 = 20$

  However, a minimal cut with capacity equal to the max flow (12) is:

  $$\text{Cut: } (a \to c), (b \to d)$$

  Cut capacity: $4 + 8 = \boxed{12}$

- **Is the Maximal Flow Unique?**

  No, the maximal flow is not unique. Ford-Fulkerson's output depends on the order of augmenting paths chosen. Multiple distributions of flow can result in the same total max flow, but with different edge-level allocations.

10.2 Exploration

This exercise highlights the fundamental principles of network flow optimization. The Ford-Fulkerson algorithm exemplifies how local decisions (augmenting paths) contribute to a global solution (maximum flow). The fact that different augmenting path orders can yield different valid flows demonstrates both the flexibility and non-determinism in algorithmic execution.

Understanding the max-flow min-cut theorem provides a dual perspective on flow problems: it connects how much "can be pushed" through a network to how the network can be "cut" to stop all flow. This duality is foundational in operations research, logistics, and even data science when modeling constraints and bottlenecks in systems.

10.3 Question

In real-world networks with dynamic capacities or multiple commodities, how does the concept of maximum flow adapt, and what algorithms extend Ford-Fulkerson to support these more complex scenarios?

11 Synthesis

Beyond the Code: Understanding the Essence of Algorithms

In computer science, I was easily caught up in implementation details: loops, conditionals, and syntax quirks, forgetting to step back and understand the deeper structure that makes an algorithm powerful or elegant. Algorithm analysis offers a perspective in which we can determine not just what the code does, but how well it performs, and why certain approaches are more efficient than others, and find the trade-offs beneath the surface of it. Through this course and personal research, I've realized that the important essence of algorithms doesn't lie in the code, but in the reasoning behind the code.

In general, algorithm analysis is about modeling problems and predicting behavior. For example, there is more than mathematical abstraction for Big-O notation; it turns into a language for describing how solutions scale as data expands. Whether comparing mergesort: $O(n \log n)$ to bubble sort: $O(n^2)$, or studying polynomial versus exponential runtimes in dynamic programming, complexity makes us think like engineers, prioritizing feasibility, cost, and performance under criteria and constraints.

One important realization for me was that algorithm analysis is not just theoretical. It's a bridge between computation and real-world systems. For example, like our max-flow problem sets, we used the Ford-Fulkerson algorithm to not just compute numbers, but to also model throughput, bottlenecks, and capacity, which I discovered are concepts that apply in everything from internet traffic routing to supply chain logistics. The abstraction of graphs and flows helped reveal to me how simple rules, when layered thoughtfully, can show surprisingly powerful solutions.

Additionally, analysis pushed for a deeper thought process and intellectual discipline. It forces us to ask hard questions, like: *Is this the most efficient way? What assumptions am I making? What happens in the worst case?* These questions are not just about runtime—they are reflections of good problem-solving habits. In this sense, algorithm analysis cultivates a mindset, not just a skillset.

To conclude, going beyond the code means knowing that code is like a vehicle, and not the destination. Behind every efficient implementation is a conceptual model built on logic, mathematics, and abstraction. To understand the essence of algorithms is to value the elegance of this invisible structure surrounding it, where clarity, precision, and strategy are tied into one. As I continue to study programming languages and systems, I now carry with me not just the ability to write better code, but to think more effectively about the problems that code is meant to solve.

12 Participation

Discord Questions

References

[ALG] Algorithm Analysis, Chapman University, 2025.

[Sip] Michael Sipser, *Introduction to the Theory of Computation*, Cengage Learning, 3rd ed., 2012.

[CLRS] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein, *Introduction to Algorithms*, MIT Press, 3rd ed., 2009.

[DM] Kenneth H. Rosen, *Discrete Mathematics and Its Applications*, McGraw-Hill, 7th ed., 2012.

[AI] Stuart Russell and Peter Norvig, *Artificial Intelligence: A Modern Approach*, Pearson, 4th ed., 2020.

[ITALC] J.E. Hopcraft, *Introduction to Automata Theory, Language, and Computation*, Pearson, 2006.