CPSC-406 Report

Ryan Benner
Chapman University

May 25, 2025

Abstract

This report is an in depth analysis and exploration of the foundational topics surrounding computational theory and algorithm analysis that were covered throughout the semester long course. These topics include things such as deterministic and non deterministic finite automata, regular expressions, formal languages, complexity analysis, Turing Machines, and decidability. The report explores the Halting problem in particular depth, and discusses the practical implications of the Halting problem and various other theories. It discusses the real life impacts of these theories on the practical field of software engineering, particularly focusing on both the challenges as well as the strategies associated with algorithm efficiency, computational limits, and how these manifest in real world applications. Finally, there is ample reflection upon the importance of these concepts, their relevance to modern day software development practices, as well as potential improvements upon how practices may evolve in the future due to rapid advancements in technology.

Contents

1 Introduction **2**

2 Week by Week **2**
2.1 Week 1 . . . 2
2.2 Week 2 . . . 2
2.3 Week 3 . . . 3
2.4 Week 4 . . . 6
2.5 Week 5 . . . 8
2.6 Week 6 . . . 10
2.7 Week 7 . . . 10
2.8 Week 8 . . . 15
2.9 Week 9 . . . 15
2.10 Week 10 . . . 17
2.11 Week 11/12 . . . 19
2.12 Week 13/14 . . . 21

3 Synthesis - Turing Machines and the Halting Problem **23**

4 Evidence of Participation **23**

5 Conclusion **26**

1 Introduction

This report takes an in depth look at the main topics covered throughout the algorithm analysis course, and connects them to real-world software engineering practices. Throughout the semester, we worked with deterministic and non-deterministic finite automata, gaining an understanding of how they influence decision making in tasks like parsing and language recognition. Next, we looked at regular expressions and formal language theory, which provided methods for recognizing patterns in data and building complex systems such as compilers. As the course progressed, more advanced topics were explored, such as Turing machines, complexity analysis, and theoretical limits of computation, with the Halting Problem being a key example of an undecidable problem.

One of the most important takeaways from the course was the realization that not everything is solvable, even in theory. Learning about the Halting Problem made it very clear to me that there are boundaries to what we can expect algorithms to accomplish, and it is important to be able to recognize this distinction to better focus attention to problems that can be solved. This has real implications in software engineering, especially in testing and verification, where developers can identify the kinds of problems that theory says can't be solved by a general algorithm. We looked at how formal methods and automated testing tools attempt to work around these issues, even if they aren't able to eliminate them entirely. Overall, the report explores abstract theory and how it relates to practical tools and thinking patterns which are highly useful when building software. Understanding these concepts helps not only with improving algorithm efficiency, but also with understanding what kinds of problems to focus on in the first place.

2 Week by Week

2.1 Week 1

Comments and Questions

How do automata models like parking machines demonstrate the concepts of states and transitions? How does complexity theory contribute to these models?

2.2 Week 2

Notes

Chapter 2.1 of ITALC discusses the basic model of computation that demonstrates the structure and components of finite automata. It explains that a finite automaton is a simple computational machine which is composed of a set of states, including a start state, and one or more end states. As the automaton reads an input, it transitions between states based on a set of rules, the transition functions. The section uses examples with simple intuition, such as an on/off switch, to show how a system can 'remember' essential information despite its simplicity. Also, it talks about the difference between deterministic and non-deterministic models. A deterministic model is one in which each state has a unique transition for each input, and a non-deterministic model is one where multiple transitions may be possible for one input. Although these two models are very different in practice, they recognize the same set of basic regular language.

Homework 1

Introduction to Automata Theory:

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

Answer: $[5,5,5,5,5], [5,5,5,10], [5,5,10,5], [5,10,5,5], [5,10,10], [10,5,5,5], [10,5,10], [10,10,5]$

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

Answer: Any word which ends in 'pay' will result in end state being unlocked

Deterministic and Non-Deterministic Finite Automata:

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

Answer:

word	A_1	A_2
aaa	no	yes
aab	yes	no
aba	no	no
abb	no	no
baa	no	yes
bab	no	no
bba	no	no
bbb	no	no

Homework: Consider the DFA from above: Consider the paths corresponding to the words $w_1 = 0010$, $w_2 = 1101$, $w_3 = 1100$.
Answer: w_1 and w_2 both achieve the end state

Comments and Questions

Because finite automata have limited memory, their ability to recognize more complex language patterns is restricted. Will the range of what is categorized as a finite automaton broaden over time? How could we modify automata to recognize more complex language?

2.3 Week 3

Homework 2

Exercise 2, 1-3:

The file **dfa.py** implements a class representing a deterministic finite automaton(DFA) The automaton is defined by a set of states, inputs, transition functions, an initial state, and accepting states, all of which are initialized in the constructor function. The repr method offers a formatted string representation of the DFA for easy inspection. The run method processes an input word by iterating through its characters and transitions between states according to the defined rules. If an undefined transition is encountered or an invalid symbol is detected, the word is rejected. Finally, the refuse method generates a complementary DFA by changing the set of accepting states to accept words that the original DFA rejects.
The file **dfa − ex01.py** implements two DFAs, A1 and A2, and runs each through dfa.py to test them repsectively.

```
# dfa.py
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 (accepting) 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
        for symbol in w:
            if symbol not in self.Sigma:
                raise ValueError(f"Symbol {symbol} not in the DFA alphabet {self.Sigma}.")
            # look up transition state, if none is defined the word is rejected
            if (current_state, symbol) not in self.delta:
                return False
            current_state = self.delta[(current_state, symbol)]
        return current_state in self.F

    # returns a new DFA that accepts the words that the current DFA refuses and vice versa
    def refuse(self):
        new_F = self.Q - set(self.F)
        return DFA(self.Q, self.Sigma, self.delta, self.q0, new_F)

    def to_NFA(self):
        # make new transition function where each DFA transition becomes a set
        from nfa import NFA
        nfa_delta = {}
        for (state, symbol), next_state in self.delta.items():
            nfa_delta[(state, symbol)] = {next_state}
        return NFA(self.Q, self.Sigma, nfa_delta, self.q0, self.F)
```

```
# dfa_ex01.py
import dfa as 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 = {'1', '2', '3', '4'}
    Sigma1 = {'a','b'}
    delta1 = {
        ('1', 'a'): '2',
        ('1', 'b'): '4',
        ('2', 'a'): '2',
        ('2', 'b'): '3',
        ('3', 'a'): '2',
        ('3', 'b'): '2',
        ('4', 'a'): '4',
```

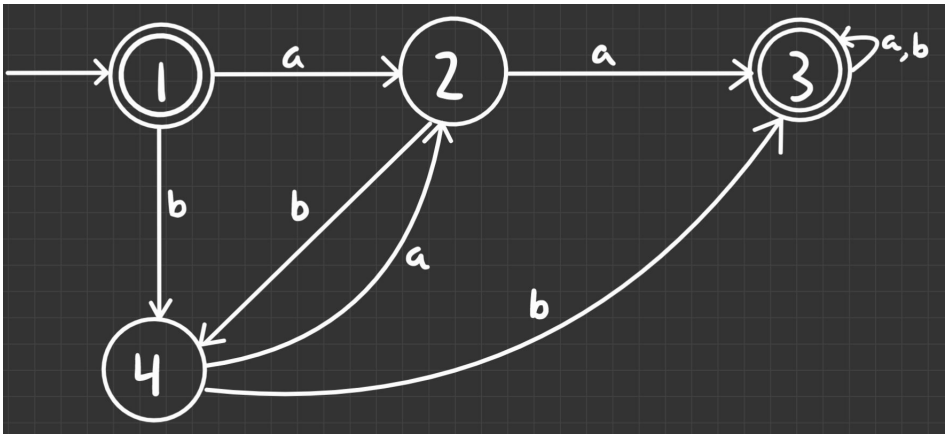


Figure 1: Lab Exercsie 4

```
    ('4', 'b'): '4'
}
q01 = '1'
F1 = {'3'}

Q2 = {'1', '2', '3'}
Sigma2 = {'a','b'}
delta2 = {
    ('1', 'a'): '2',
    ('1', 'b'): '1',
    ('2', 'a'): '3',
    ('2', 'b'): '1',
    ('3', 'a'): '3',
    ('3', 'b'): '1',
}
q02 = '1'
F2 = {'3'}

A1 = dfa.DFA(Q1, Sigma1, delta1, q01, F1)
A2 = dfa.DFA(Q2, Sigma2, delta2, q02, F2)

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

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

__main__()
```

Comments and Questions

I occasionally find myself struggling to recognize the pattern when first looking at a new DFA. Is there an insightful method or trick to recognizing these patterns more easily?

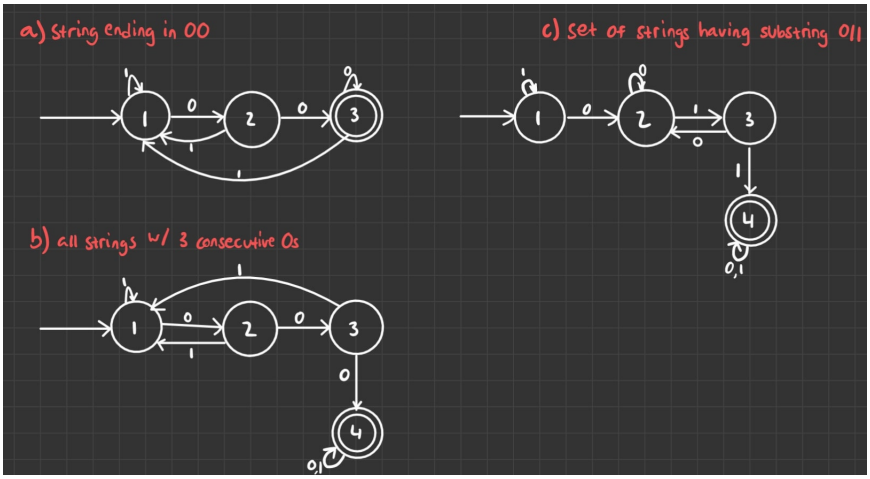


Figure 2: ITALC Exercise 2.2.4

2.4 Week 4

Homework 3

Homework 1

1. The language of the automata A2 can be described as starting with an a and having an odd length. Any b's that may occur must do so from state 2. This can be described as a regular expression $a((a|b)a)*$.

2. The extended transition functions can be evaluated as follows:

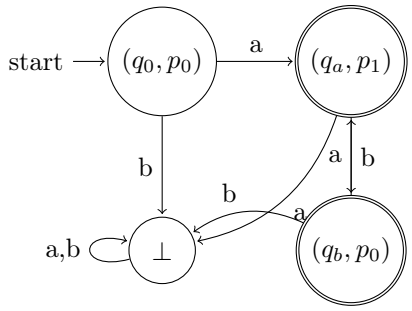
A1:

- $\delta_1(1, a) = 2$
- $\delta_1(2, b) = 4$
- $\delta_1(4, a) = 2$
- $\delta_1(2, a) = 3$

A2:

- $\delta_2(1, a) = 2$
- $\delta_2(2, b) = 1$
- $\delta_2(1, b) = 3$
- $\delta_2(3, a) = 3$

Homework 2



2. When constructing the product automaton, a word w is processed by both components simultaneously. The word is said to be accepted iff the computation ends in a state (q, p) where q is in F(1) and p is in F(2). This means that w is accepted by both A(1) and A(2), so L(A) can be the intersection between the two.

3. In the product automaton constrction, the set of states is always $Q_1 * Q_2$ and the transitions are the same. To find an automaton which is the union of the two languages, we change the accepting states. So, a word w is accepted by A' iff it is accepted by at least one of the original automata, which gives L(A') to be the union of L(A1) and L(A2).

To summarize:

$$\begin{aligned} Q &= Q^{(1)} \times Q^{(2)}, \\ \delta((q,p),x) &= (\delta^{(1)}(q,x),\, \delta^{(2)}(p,x)), \\ q_0 &= (q_0^{(1)}, q_0^{(2)}), \\ F' &= \Big\{(q,p) \in Q^{(1)} \times Q^{(2)} : q \in F^{(1)} \text{ or } p \in F^{(2)}\Big\}. \end{aligned}$$

Exercise 2.2.7 - ITALC

Theorem. Let $A = (Q, \Sigma, \delta, q_0, F)$ be a DFA and let $q \in Q$ be such that

$$\delta(q, a) = q \quad \text{for all } a \in \Sigma.$$

Then for all $w \in \Sigma^*$, $\hat{\delta}(q, w) = q$.

Proof. We prove by induction on the length of w.

Base Case: $w = \varepsilon$ (the empty string). By definition of the extended transition function,

$$\hat{\delta}(q, \varepsilon) = q.$$

Inductive Step: Assume that for some $w \in \Sigma^*$, $\hat{\delta}(q, w) = q$. Let $a \in \Sigma$. Then,

$$\hat{\delta}(q, wa) = \delta(\hat{\delta}(q, w), a) = \delta(q, a) = q.$$

Thus, by the principle of induction, for all $w \in \Sigma^*$, $\hat{\delta}(q, w) = q$.

Comments and Questions

Do automata with some form of union or intersection have real world applications relating to natural language processing? Could they be used to improve parsing in conversational AI?

2.5 Week 5

Notes

Chapter 3.1-3.2 of ITALC explores regular expressions as a powerful way to more accurately describe formal languages, specifically being useful in various text processing. Regular expressions are built using basic operations such as union, concatenation,and Kleene closure, which combines simple patterns in order to create more complex expressions. These operations and the way they are used is defined through standard set operations. Union is able to combine languages by joining their elements, concatenation makes new strings by combining parts from each language, and Kleene closure allows for the generation of all potential strings. Also, Kleene's algorithm is shown as a structured method in order to change DFAs into regular expressions, usually relying on induction that builds expressions incrementally that represent state transitions. With larger automata, this algorithm can become very computationally expensive. Finally, the transition between regular expressions and automata is bidirectional, meaning you can go from one to the other and back again.

Homework 4

Homework 1:

Let $\mathcal{A} = (Q, \Sigma, \delta : Q \times \Sigma \to Q, q_0, F)$ be a DFA.

1. **Viewing $\mathcal{A}$ as an NFA:**

   A DFA is a specific case of an NFA where the transition function returns only one next state for each input symbol and state. To show the DFA as an NFA, we make a new transition function $\delta' : Q \times \Sigma \to \mathcal{P}(Q)$ sucsoh that:

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

   This shows that each deterministic transition becomes a singleton set in the NFA transition function. All other components of the DFA remain unchanged:

   - $Q' = Q$
   - Σ remains the same
   - $q_0' = q_0$
   - $F' = F$

2. **General Construction:**

   Given a DFA $\mathcal{A} = (Q, \Sigma, \delta, q_0, F)$, we define an equivalent NFA $\mathcal{A}' = (Q', \Sigma, \delta', q_0', F')$ as follows:

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

3. **Justification:**

   This design ensures that $\mathcal{A}'$ accepts the same language as $\mathcal{A}$. Because each transition in the NFA δ' corresponds to the DFA's transition , the NFA has exactly one computational path for any given input string, mirroring the DFA. So,

   $$L(\mathcal{A}) = L(\mathcal{A}')$$

Homework 2:

1. **Language accepted by $\mathcal{A}$:** The NFA accepts all binary strings that have the substring `010`.
2. **Specification of $\mathcal{A}$:**
   - $Q = \{q_0, q_1, q_2, q_3\}$
   - $\Sigma = \{0, 1\}$
   - q_0 is the start state
   - $F = \{q_3\}$
   - Transition function δ:

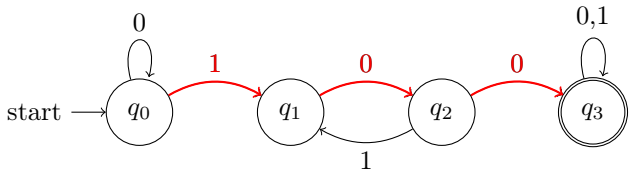
$$\begin{aligned}
&\delta(q_0, 0) = \{q_0\} \\
&\delta(q_0, 1) = \{q_0, q_1\} \\
&\delta(q_1, 0) = \{q_2\} \\
&\delta(q_2, 0) = \{q_3\}, \quad \delta(q_2, 1) = \{q_1\} \\
&\delta(q_3, 0) = \{q_3\}, \quad \delta(q_3, 1) = \{q_3\}
\end{aligned}$$

3. **Extended transition: compute $\hat{\delta}(q_0, 10110)$ step-by-step:**
   - Step 0: $\{q_0\}$
   - Read 1: $\{q_0, q_1\}$
   - Read 0: $\{q_0, q_2\}$
   - Read 1: $\{q_0, q_1\}$
   - Read 1: $\{q_0, q_1\}$
   - Read 0: $\{q_0, q_2\}$
   - Final result: $\hat{\delta}(q_0, 10110) = \{q_0, q_2\}$
4. **All paths for $v = 1100$ and $w = 1010$:**

   Step-by-step for $v = 1100$:
   - Start at $\{q_0\}$
   - Read 1: $\{q_0, q_1\}$
   - Read 1: $\{q_0, q_1\}$
   - Read 0: $\{q_0, q_2\}$
   - Read 0: $\{q_0, q_3\} \rightarrow$ accepted

   Step-by-step for $w = 1010$:
   - Start at $\{q_0\}$
   - Read 1: $\{q_0, q_1\}$
   - Read 0: $\{q_0, q_2\}$
   - Read 1: $\{q_0, q_1\}$
   - Read 0: $\{q_0, q_2\} \rightarrow$ rejected

   Common diagram of all paths:



5. **Powerset construction for DFA $\mathcal{A}^D$:**

   Start from state $\{q_0\}$:

   - $\delta(\{q_0\}, 0) = \{q_0\}$
   - $\delta(\{q_0\}, 1) = \{q_0, q_1\}$

   Continue expanding:

   - $\delta(\{q_0, q_1\}, 0) = \{q_0, q_2\}$
   - $\delta(\{q_0, q_1\}, 1) = \{q_0, q_1\}$
   - $\delta(\{q_0, q_2\}, 0) = \{q_0, q_3\}$
   - $\delta(\{q_0, q_2\}, 1) = \{q_0, q_1\}$
   - $\delta(\{q_0, q_3\}, 0) = \{q_0, q_3\}$, etc.

   Accepting states: all subsets that contain q_3

6. **Verification and minimization:** $L(\mathcal{A}) = L(\mathcal{A}^D)$ by construction. A simple DFA for the language contains substring '010' has 4 states, so $\mathcal{A}^D$ can be simplified further.

Comments and Questions

How does the powerset construction in Homework 2 show the relationship between DFAs and NFAs, and how does this relate to the transformation approach in Homework 1?

2.6 Week 6

Comments and Questions

Why does state equivalence require all the suffixes, which might be infinite? Is there a bound on suffix length past which no new differences between states can arise?

2.7 Week 7

Homework 5

DFA Regular Expressions and State Elimination (Exercise 3.2.1)

Given a transition table:

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

a. $R_{ij}^{(0)}$ (Initial regular expressions with no intermediate states)

Let

$$R_{ij}^{(0)} = \begin{cases} \text{the symbol(s) labeling direct transitions from state } i \text{ to state } j, \\ \varnothing, \quad \text{if no direct transition,} \\ \epsilon, \quad \text{if } i = j. \end{cases}$$

Thus,

$$\begin{aligned} R_{11}^{(0)} &= \epsilon, \quad R_{12}^{(0)} = 0, \quad R_{13}^{(0)} = \varnothing, \\ R_{21}^{(0)} &= 1, \quad R_{22}^{(0)} = \epsilon, \quad R_{23}^{(0)} = 0, \\ R_{31}^{(0)} &= \varnothing, \quad R_{32}^{(0)} = 1, \quad R_{33}^{(0)} = 0 \mid \epsilon. \end{aligned}$$

b. $R_{ij}^{(1)}$ (Using state q_1 as intermediate)

Using the formula

$$R_{ij}^{(k)} = R_{ij}^{(k-1)} \cup R_{ik}^{(k-1)} \left(R_{kk}^{(k-1)}\right)^* R_{kj}^{(k-1)},$$

with q_1 as the singular intermediate state we get:

$$\begin{aligned} R_{11}^{(1)} &= \epsilon \cup (\epsilon)(\epsilon)^*\epsilon = \epsilon, \\ R_{12}^{(1)} &= 0 \cup (\epsilon)(\epsilon)0 = 0, \\ R_{13}^{(1)} &= \varnothing \cup (\epsilon)(\epsilon)\varnothing = \varnothing, \\ R_{21}^{(1)} &= 1 \cup (1)(\epsilon)^*\epsilon = 1, \\ R_{22}^{(1)} &= \epsilon \cup (1)(\epsilon)0 = 10, \\ R_{23}^{(1)} &= 0 \cup (1)(\epsilon)\varnothing = 0, \\ R_{31}^{(1)} &= \varnothing \cup (1)(\epsilon)^*\epsilon = \varnothing, \\ R_{32}^{(1)} &= 1 \cup (1)(\epsilon)0 = 1, \\ R_{33}^{(1)} &= 0 \cup (1)(\epsilon)\varnothing = 0. \end{aligned}$$

c. $R_{ij}^{(2)}$ (Using q_1 and q_2 as intermediates)

Now we add q_2 as an intermediate state:

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

d. Regular Expression for the Language

Start state: q_1 Final state: q_3.

The regex is $R_{13}^{(2)}$, so

$$R = 0\,(10)^*\,0.$$

e. Transition Diagram and State Elimination (eliminate q_2)

A simpler diagram of the DFA is:

$$q_1 \xrightarrow{0} q_2 \xrightarrow{0} q_3 \quad \text{(final)}$$

with loops and additional transitions that are indicated in the original DFA. After getting rid of q_2, the path from q_1 to q_3 becomes:

$$(1 \cup 01)00(0),$$

so final regular expression becomes

$$R = (1 \cup 01)00(0).$$

DFA Regular Expressions and State Elimination (Exercise 3.2.2)

Given the transition table:

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

a. $R_{ij}^{(0)}$ (Initial regular expressions with no intermediate states)

Let

$$\begin{aligned} R_{11}^{(0)} &= \epsilon, \quad R_{12}^{(0)} = 0, \quad R_{13}^{(0)} = 1, \\ R_{21}^{(0)} &= 0, \quad R_{22}^{(0)} = \epsilon, \quad R_{23}^{(0)} = 1, \\ R_{31}^{(0)} &= 1, \quad R_{32}^{(0)} = 0, \quad R_{33}^{(0)} = \epsilon. \end{aligned}$$

b. $R_{ij}^{(1)}$ (Using q_1 as intermediate)

Apply:

$$R_{ij}^{(1)} = R_{ij}^{(0)} \cup R_{i1}^{(0)}\,(R_{11}^{(0)})^*\,R_{1j}^{(0)}.$$

This gives:

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

c. $R_{ij}^{(2)}$ (Using q_1 and q_2 as intermediates)

Apply:

$$R_{ij}^{(2)} = R_{ij}^{(1)} \cup R_{i2}^{(1)} \left(R_{22}^{(1)}\right)^* R_{2j}^{(1)}.$$

For example:

$$\begin{aligned} R_{11}^{(2)} &= \epsilon \cup 0\left((\epsilon \cup 00)\right)^* 0, \\ R_{12}^{(2)} &= 0 \cup 0\left((\epsilon \cup 00)\right)^* (\epsilon \cup 00), \\ R_{13}^{(2)} &= 1 \cup 0\left((\epsilon \cup 00)\right)^* (1 \cup 01), \end{aligned}$$

and in a similar manner for the other entries.

d. Regular Expression for the Language

Start state: q_1 Final state: q_3.

Thus,

$$R = R_{13}^{(2)} = 1 \cup 0\left((\epsilon \cup 00)\right)^* (1 \cup 01).$$

e. Transition Diagram and State Elimination (eliminate q_2)

A simplified diagram is as follows:

$$q_1 \xrightarrow{0} q_2 \xrightarrow{0} q_1 \quad \text{and} \quad q_3 \xleftarrow{1} q_3.$$

By getting rid of q_2 and considering the other paths:

$$q_1 \xrightarrow{0} q_2 \xrightarrow{1} q_3 \quad \Rightarrow \quad 01,$$

$$q_1 \xrightarrow{0} q_2 \xrightarrow{0} q_1 \quad \Rightarrow \quad 00 \text{ (loop)},$$

the final regex becomes:

$$R = (1 \cup 0(00)^*1).$$

DFA Minimization (Exercise 4.4.1)

Given the 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. Table of Distinguishabilities

Final table (X = marked/inequivalent, blank = equivalent): The unmarked (equivalent) states are E, F, and G.

b. Minimum-State Equivalent DFA Define the minimized DFA $M' = (Q', \Sigma, \delta', q_0', F')$ as follows:

	B	C	D	E	F	G	H
A	X	X	X	X	X		X
B		X	X	X		X	X
C			X		X	X	X
D				X	X	X	X
E					X	X	X
F						X	X
G							X
H							

Table 1: Table of distinguishable state pairs

$$
\begin{aligned}
Q' &= \{\, [AG],\, [BF],\, [CE],\, [D],\, [H] \,\}, \\
\Sigma &= \{0, 1\}, \\
q_0' &= [AG], \\
F' &= \{[D]\}.
\end{aligned}
$$

The transition function δ' is given by:

δ'	0	1
[AG]	[BF]	[AG]
[BF]	[AG]	[CE]
[CE]	[D]	[BF]
[D]	[D]	[AG]
[H]	[AG]	[D]

DFA Minimization (Exercise 4.4.2)

Given the transition table:

	0	1
$\rightarrow A$	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. Table of Distinguishabilities

Final states: C, F, and I.

Mark all the pairs with one final and one non-final state, then make distinctions. The final table (X = distinguishable) is:

b. Minimum-State Equivalent DFA

	B	C	D	E	F	G	H	I
A	X	X	X	X	X	X	X	X
B		X	X	X	X	X	X	X
C			X	X	X	X	X	X
D				X	X	X	X	X
E					X	X	X	X
F						X	X	X
G							X	X
H								X

Table 2: Table of distinguishable state pairs

Define $M = (Q, \Sigma, \delta, q_0, F)$ where

$$\begin{aligned} Q &= \{[\text{A}], [\text{B}], [\text{C}], [\text{D}], [\text{E}], [\text{F}], [\text{G}], [\text{H}], [\text{I}]\}, \\ \Sigma &= \{0, 1\}, \\ q_0 &= [\text{A}], \\ F &= \{[\text{C}], [\text{F}], [\text{I}]\}. \end{aligned}$$

The transition function δ is given by:

δ	0	1
[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]

Comments and Questions

In Exercise 3.2.1, how does the order of state elimination change the final regular expression's structure, and could a different order get us to a more simplified form?

2.8 Week 8

Comments and Questions

Suppose you have a recursively enumerable language L accepted by a TM M. What conditions are needed to guarantee there is a halting decider M' for L, and how would you go about constructing such an M'?

2.9 Week 9

Homework 6

Exercise A (1). TM for $L = \{1\,0^n : n \in \mathbb{N}\}$ that outputs $1\,0^{n+1}$:

$$M_1 = \big(Q_1, \Sigma, \Gamma, \delta_1, q_0, B, F_1\big), \quad Q_1 = \{q_0, q_1, q_{\text{acc}}, q_{\text{rej}}\}, \quad \Sigma = \{0, 1\},\ \Gamma = \{0, 1, B\},\ F_1 = \{q_{\text{acc}}\}.$$

$$\delta_1 \colon Q_1 \times \Gamma \to \Gamma \times \{L, R, S\} \times Q_1$$

$$\begin{aligned}\delta_1(q_0,1) &= (1,R,q_1), \quad \delta_1(q_0,0) = (\_,\_,q_{\text{rej}}),\\ \delta_1(q_0,B) &= (\_,\_,q_{\text{rej}}), \quad \delta_1(q_1,0) = (0,R,q_1),\\ \delta_1(q_1,B) &= (0,S,q_{\text{acc}}).\end{aligned}$$

Exercise A (2). TM for $L = \{1\,0^n\}$ that outputs just "1":

$$M_2 = \big(Q_2,\Sigma,\Gamma,\delta_2,q_0,B,F_2\big), \quad Q_2 = \{q_0,q_1,q_2,q_{\text{acc}},q_{\text{rej}}\}, \quad \Sigma = \{0,1\},\ \Gamma = \{0,1,B\},\ F_2 = \{q_{\text{acc}}\}.$$

$$\delta_2 \colon Q_2 \times \Gamma \to \Gamma \times \{L,R,S\} \times Q_2$$

$$\begin{aligned}\delta_2(q_0,1) &= (1,R,q_1), \quad \delta_2(q_0,0) = (\_,\_,q_{\text{rej}}),\\ \delta_2(q_0,B) &= (\_,\_,q_{\text{rej}}), \quad \delta_2(q_1,0) = (0,R,q_1),\\ \delta_2(q_1,B) &= (B,L,q_2), \quad \delta_2(q_2,0) = (B,L,q_2),\\ \delta_2(q_2,1) &= (1,S,q_{\text{acc}}), \quad \delta_2(q_2,B) = (\_,\_,q_{\text{rej}}).\end{aligned}$$

Exercise A (3). TM for all binary strings that swaps $0 \leftrightarrow 1$:

$$M_3 = \big(Q_3,\Sigma,\Gamma,\delta_3,q_0,B,F_3\big), \quad Q_3 = \{q_0,q_{\text{acc}}\}, \quad \Sigma = \{0,1\},\ \Gamma = \{0,1,B\},\ F_3 = \{q_{\text{acc}}\}.$$

$$\delta_3 \colon Q_3 \times \Gamma \to \Gamma \times \{L,R,S\} \times Q_3$$

$$\begin{aligned}\delta_3(q_0,0) &= (1,R,q_0), \quad \delta_3(q_0,1) = (0,R,q_0),\\ \delta_3(q_0,B) &= (B,S,q_{\text{acc}}).\end{aligned}$$

Homework 7

Exercise 1.1
Let

$$L_1 = \{M \mid M \text{ halts on } \langle M\rangle\}.$$

L_1 is recursively enumerable but not decidable, and its complement is not recursively enumerable.

Exercise 1.2
Let

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

L_2 is recursively enumerable but not decidable, and its complement is not recursively enumerable.

Exercise 1.3
Let

$$L_3 = \{(M,w,k) \mid M \text{ halts on } w \text{ within } k \text{ steps}\}.$$

L_3 is decidable (hence both recursively enumerable and co-recursively enumerable).

Exercise 2.1. True. Argument: If M_1 and M_2 decide L_1 and L_2, then when you have an input w run $M_1(w)$; if it accepts, accept; otherwise run $M_2(w)$; if it accepts, accept; else it rejects. This always halts and decides if $L_1 \cup L_2$.

Exercise 2.2. True. Argument: If M decides L, then when you have an input w run $M(w)$ and flip the output (accept↔reject). The resulting machine will halt on every input and decides $\overline{L}$.

Exercise 2.3. True. Argument: To decide L^*, on an input w use dynamic programming principles. Set $\text{dp}[0] = \text{true}$ and for $1 \le i \le |w|$, let:

$$\text{dp}[i] = \exists\, j < i \ \big[\text{dp}[j] \wedge M\big(w_{j+1} \dots w_i\big) \text{ accepts}\big].$$

Then accept iff $\text{dp}[|w|]$. This halts in a finite amount of steps.

Exercise 2.4. True. Argument: If M_1 and M_2 recognize L_1 and L_2, combine their simulations on input w. If either accepts, accept. This kind of decides $L_1 \cup L_2$, so $L_1 \cup L_2$ is r.e.

Exercise 2.5. False. Counterexample: The halting set

$$K = \{\langle M, w\rangle \mid M \text{ halts on } w\}$$

is r.e. but its complement $\overline{K}$ is not r.e. So, r.e. languages are not closed under the complement.

Exercise 2.6. True. Argument: To recognize L^*, we nondeterministically split $w = w_1 \cdots w_k$, 'dovetail'-run the recognizer M on each w_i, and accept if all accept. Every $w \in L^*$ will have some accepting 'decomposition', so this semi-decides L^*, but not completely.

Comments and Questions

1. What modifications are required to combine the "increment-zeros" TM (Ex. 1) and the "bit-swap" TM (Ex. 3) into a single 2-tape TM that performs both operations in one pass? 2. Is there a way we can augment the standard TM model that would greatly enlarge the overall class of decidable languages?

2.10 Week 10

Notes

input len n, interested in num of steps for n also interested in maximal number of stepss T: N -> R >= 0 looking at non negative functions from f,g:N -> R >= 0 we say the f is an element of O(g) iff f(n) <= M * g(n), we want this to hold true for some M > 0, and all n > N this is a measure with which we can compare the growth of functions

Homework 8 and 9

Due April 27th

Exercise 1 (Warmup)

$$\begin{aligned}
&f. \quad \log\log n,\\
&c. \quad \log n,\\
&b. \quad e^{\log n} = n,\\
&e. \quad e^{2\log n} = n^2,\\
&g. \quad 2^n,\\
&d. \quad e^n,\\
&h. \quad n!,\\
&a. \quad 2^{2^n}.
\end{aligned}$$

Exercise 2

Exercise 2.1 *Claim:* $f \in O(f)$. *Proof.* Choose $c = 1$ and any n_0. Then for all $n \geq n_0$,

$$f(n) \leq 1 \cdot f(n).$$

Exercise 2.2 *Claim:* If $c > 0$ is constant then $O(c \cdot f) = O(f)$. *Proof.*

$\subseteq$ If $g \in O(c\,f)$ then there exist constants C, n_1 such that for all $n \geq n_1$,

$$g(n) \leq C\,(c\,f(n)) = (C\,c)\,f(n),$$

so $g \in O(f)$.

$\supseteq$ If $g \in O(f)$ then there exist constants D, n_2 such that for all $n \geq n_2$,

$$g(n) \leq D\,f(n) = (D/c)\,(c\,f(n)),$$

so $g \in O(c\,f)$.

Exercise 2.3 *Claim:* If $f(n) \leq g(n)$ for all $n \geq n_0$, then $O(f) \subseteq O(g)$. *Proof.* Let $h \in O(f)$, so there exist C, n_1 such that for all $n \geq n_1$,

$$h(n) \leq C\,f(n) \leq C\,g(n).$$

Thus $h \in O(g)$.

Exercise 2.4 *Claim:* If $O(f) \subseteq O(g)$ then $O(f+h) \subseteq O(g+h)$. *Proof.* From $O(f) \subseteq O(g)$ we have $f \in O(g)$, so there exist A, N such that for all $n \geq N$,

$$f(n) \leq A\,g(n).$$

If $k \in O(f+h)$ then there exist C, n_0 such that for all $n \geq n_0$,

$$k(n) \leq C\big(f(n)+h(n)\big) \leq C\big(A\,g(n)+h(n)\big) \leq C\max(A,1)\,\big(g(n)+h(n)\big),$$

hence $k \in O(g+h)$.

Exercise 2.5 *Claim:* If $h(n) > 0$ for all n and $O(f) \subseteq O(g)$ then $O(f \cdot h) \subseteq O(g \cdot h)$. *Proof.* Again $f \in O(g)$, so there exist A, N such that for all $n \geq N$,

$$f(n) \leq A\,g(n).$$

If $k \in O(f \cdot h)$ then there exist C, n_0 such that for all $n \geq n_0$,

$$k(n) \leq C\,f(n)\,h(n) \leq C\,(A\,g(n))\,h(n) = (C\,A)\,\big(g(n)\,h(n)\big),$$

hence $k \in O(g \cdot h)$.

Exercise 3

Exercise 3.1 *Claim:* If $j \leq k$ then $O(n^j) \subseteq O(n^k)$. *Proof.* For all $n \geq 1$, $n^j \leq n^k$. If $f \in O(n^j)$ then there exist C, n_0 such that for all $n \geq n_0$,

$$f(n) \;\leq\; C\,n^j \;\leq\; C\,n^k,$$

so $f \in O(n^k)$.

Exercise 3.2 *Claim:* If $j \leq k$ then $O(n^j + n^k) \subseteq O(n^k)$. *Proof.* For all $n \geq 1$, $n^j + n^k \leq 2\,n^k$. If $g \in O(n^j + n^k)$ then there exist C, n_0 such that for all $n \geq n_0$,

$$g(n) \;\leq\; C\,(n^j + n^k) \;\leq\; 2C\,n^k,$$

so $g \in O(n^k)$.

Exercise 3.3 *Claim:* $O\big(\sum_{i=0}^{k} a_i\,n^i\big) = O(n^k)$. *Proof.* Let $A = \sum_{i=0}^{k} |a_i|$. For all $n \geq 1$,

$$\sum_{i=0}^{k} a_i\,n^i \;\leq\; \sum_{i=0}^{k} |a_i|\,n^k \;=\; A\,n^k.$$

Hence any $h \in O\big(\sum a_i\, n^i\big)$ satisfies $h(n) \le C\, A\, n^k$ for some C, n_0, so $h \in O(n^k)$.

Exercise 3.4 *Claim:* $O(\log n) \subseteq O(n)$. *Proof.* For all $n \ge 2$, $\log n \le n$. If $p \in O(\log n)$ then there exist C, n_0 such that for all $n \ge \max(2, n_0)$,

$$p(n) \;\le\; C\, \log n \;\le\; C\, n,$$

so $p \in O(n)$.

Exercise 3.5 *Claim:* $O(n \log n) \subseteq O(n^2)$. *Proof.* For all $n \ge 2$, $\log n \le n$, hence $n \log n \le n^2$. If $q \in O(n \log n)$ then there exist C, n_0 such that for all $n \ge \max(2, n_0)$,

$$q(n) \;\le\; C\, n \log n \;\le\; C\, n^2,$$

so $q \in O(n^2)$.

Comments and Questions

Why does Big-O ignore constant factors and smaller terms, and how does that help us choose and improve algorithms?

2.11 Week 11/12

Homework 10/11

Exercise 1

1.

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

$$\begin{aligned}
&= \neg(a \wedge b) \;\wedge\; \neg(\neg c \wedge d) && \text{(De Morgan)}\\
&= (\neg a \vee \neg b) \;\wedge\; (\neg\neg c \vee \neg d) && \text{(De Morgan)}\\
&= (\neg a \vee \neg b) \;\wedge\; (c \vee \neg d)
\end{aligned}$$

2.

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

$$\begin{aligned}
&= \neg\big(\neg(p \vee q) \;\vee\; (r \wedge \neg s)\big) && \text{(implication } A \to B \equiv \neg A \vee B)\\
&= \neg\neg(p \vee q) \;\wedge\; \neg(r \wedge \neg s) && \text{(De Morgan)}\\
&= (p \vee q) \;\wedge\; (\neg r \vee s)
\end{aligned}$$

Exercise 2

$\psi_1 = (a \vee \neg b) \wedge (\neg a \vee b) \wedge (\neg a \vee \neg b)$
Try $a = 0:\ (a \vee \neg b) \Rightarrow b = 0,\ (\neg a \vee b) = 1 \implies (a, b) = (0, 0)$ satisfies all.

$\psi_2 = (\neg p \vee q) \wedge (\neg q \vee r) \wedge \neg(\neg p \vee r)$
$\equiv (\neg p \vee q) \wedge (\neg q \vee r) \wedge (p \wedge \neg r)$
$\Rightarrow q = 1,\ (\neg q \vee r) = 0 \implies$ unsatisfiable.

$\psi_3 = (x \vee y) \wedge (\neg x \vee y) \wedge (x \vee \neg y) \wedge (\neg x \vee \neg y)$
$\implies x \iff y$
$\{x, y\} = \{0, 0\}$ fails $(x \vee y)$, $\{1, 1\}$ fails $(\neg x \vee \neg y) \implies$ unsatisfiable.

Exercise 3 Sudoku

Let

$$x_{r,c,v} \;=\; \begin{cases} \text{true}, & \text{if the cell in row } r \text{ and column } c \text{ holds the digit } v, \\ \text{false}, & \text{otherwise,} \end{cases}$$

for $r, c, v \in \{1, \ldots, 9\}$. Our overall formula is

$$\Phi \;=\; C_1 \;\wedge\; C_2 \;\wedge\; C_3 \;\wedge\; C_4 \;\wedge\; C_5 \;\wedge\; C_6\,.$$

Constraint 1: Each cell has at least one digit

$$C_1 \;=\; \bigwedge_{r=1}^{9} \bigwedge_{c=1}^{9} \Big(\bigvee_{v=1}^{9} x_{r,c,v} \Big)$$

Constraint 2: No cell contains two different digits

$$C_2 \;=\; \bigwedge_{r=1}^{9} \bigwedge_{c=1}^{9} \bigwedge_{1 \le v < v' \le 9} \big(\neg x_{r,c,v} \;\vee\; \neg x_{r,c,v'} \big)$$

Constraint 3: Each digit appears in every row

$$C_3 \;=\; \bigwedge_{r=1}^{9} \bigwedge_{v=1}^{9} \Big(\bigvee_{c=1}^{9} x_{r,c,v} \Big)$$

Constraint 4: Each digit appears in every column

$$C_4 \;=\; \bigwedge_{c=1}^{9} \bigwedge_{v=1}^{9} \Big(\bigvee_{r=1}^{9} x_{r,c,v} \Big)$$

Constraint 5: Each digit appears in each 3×3 block

Index blocks by $b_r, b_c \in \{0, 1, 2\}$. So

$$C_5 \;=\; \bigwedge_{v=1}^{9} \bigwedge_{b_r=0}^{2} \bigwedge_{b_c=0}^{2} \Big(\bigvee_{r=3b_r+1}^{3b_r+3} \bigvee_{c=3b_c+1}^{3b_c+3} x_{r,c,v} \Big)$$

Constraint 6: Respect the given clues

The preset entries will be $G = \{(r_i, c_i, v_i) \mid i = 1, \ldots, m\}$. So

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

Comments and Questions

Is this the most efficient way to encode a problem like Sudoku? Are there other algorithmic approaches that are better or worse to solving such a problem?

2.12 Week 13/14

Homework 12/13

Exercise 1

1. Applying Ford–Fulkerson algorithm:

$$\begin{aligned}&\text{Augment } s \to a \to d \to t \text{ by } 8,\\ &\text{Augment } s \to b \to d \to t \text{ by } 2,\\ &\text{Augment } s \to a \to c \to t \text{ by } 2,\\ &\text{Augment } s \to b \to d \to c \to t \text{ by } 6,\\ &\text{Augment } s \to b \to d \to a \to c \to t \text{ by } 1.\end{aligned}$$

Resulting edge flows:

$$f(s,a) = 10, \quad f(s,b) = 9, \quad f(a,c) = 3, \quad f(a,d) = 7, \quad f(b,d) = 9, \quad f(d,c) = 6, \quad f(c,t) = 9, \quad f(d,t) = 10.$$

So, total flow is

$$|f| \;=\; f(s,a) + f(s,b) \;=\; 10 + 9 \;=\; 19.$$

2. In the final residual graph, the set of vertices reachable from s is

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

The capacity of the cut (S, T) is

$$\text{cap}(s \to a) \;+\; \text{cap}(b \to d) \;=\; 10 + 9 = 19,$$

so, its a minimum cut.

3. A maximum flow doesn't need to be unique. Flow can be rerouted along directed cycles in the residual graph to get different edge-flow assignments with same total value.

Exercise 2

1. The algorithm computes

$$r \;=\; \sum_{k=1}^{n-1} \sum_{l=k+1}^{n} \sum_{m=1}^{l} 1 \;=\; \sum_{k=1}^{n-1} \sum_{l=k+1}^{n} l \;=\; \sum_{l=2}^{n} l\,(l-1) \;=\; \sum_{l=1}^{n} (l^2 - l) \;=\; \sum_{l=1}^{n} l^2 \;-\; \sum_{l=1}^{n} l.$$

Using the identities

$$\sum_{l=1}^{n} l \;=\; \frac{n(n+1)}{2}, \qquad \sum_{l=1}^{n} l^2 \;=\; \frac{n(n+1)(2n+1)}{6},$$

we obtain

$$r = \frac{n(n+1)(2n+1)}{6} - \frac{n(n+1)}{2} = \frac{n(n+1)\big[(2n+1) - 3\big]}{6} = \frac{n(n+1)\,2(n-1)}{6} = \frac{n(n+1)(n-1)}{3} = \frac{n^3 - n}{3}.$$

2. The total number of constant-time increments is

$$\sum_{k=1}^{n-1} \sum_{l=k+1}^{n} \sum_{m=1}^{l} 1 = \Theta(n^3),$$

so the worst-case running time is

$$O(n^3).$$

Comments and Questions

How do different augmenting paths change the residual graph, and can they have multiple minimum cuts with the same capacity?

3 Synthesis - Turing Machines and the Halting Problem

In computational theory, one very important model is the concept of a Turing Machine [ALG] [ITALC]. These models are very abstract in nature, and are capable of performing a massive variety of tasks through the manipulation of different symbols on a 'tape' according to a certain set of rules, both of which can vary based on the machine's application. This machine was initially introduced in 1936 by Alan Turing, who is prominently described as the grandfather of computing [ITALC]. Because of their large array of applications, Turing machines have become the primary framework for understanding the computation of algorithms, and have also become central to theoretical computer science across a variety of disciplines. These machines represent the idea of computability, and are able to define the boundaries of what complex problems are solvable by an algorithm, and which ones are simply beyond being solved by an algorithm [ITALC].

One such problem, the Halting Problem, which is largely associated with Turing machines, seeks to determine through an algorithm whether or not it is possible that a given Turing machine will halt when given a specific input, or if it will continue to run indefinitely until shut down by an outside source [ITALC]. Through a technique called diagonalization, Alan Turing was able to prove that the Halting Problem is undecidable, which means that there is no general algorithm that is capable of determining whether this problem can be solved for all possible machine-input pairs. This is a very important discovery in the field of theoretical computation - The fact that it is undecidable indicates that there is a major limitation in the theory of computation, and demonstrates that while many problems can be solved through an algorithm and classified as decidable, there are many problem still that are impossible to solve via an algorithmic approach.

The fact that the Halting Problem is undecidable carries quite significant implications in the real world, particularly in software engineering debugging and testing processes. Because there isn't a universal way to determine through an algorithm whether or not a piece of software will halt or run forever, software engineers have to rely on other techniques to make these determinations on their own. Some of the methods that are utilized to test software and decidability include heuristics, approximations, and various unit tests that are designed specifically for that piece of software [ITALC]. Unfortunately, these limitations lead to unreliability and the software may only be as good as its test, which is purely reliant on the skill and experience of those designing the testing methods. This calls for very careful design strategies, thorough testing methods, and being able to apply more formal verification methods when possible in order to proactively identify and solve potential problems that may arise with undecidable software. For example, in industry, automated testing frameworks are very common practice, which are able to periodically use algorithms and different code analysis tools to identify problematic code behavior which may not be apparent to those designing the software. These tools are the most modern and sophisticated way of ensuring that software is up to standards and will not run into the challenges of undecidability, but they still are not 100

Turing machines and the Halting Problem have not only pushed computing past previous theoretical limits, but also have real world implications upon both theoretical and practical applications of computing. This exploration has guided computer scientists to better understand the differences between solving problems efficiently, and those problems that are massively computationally intensive, or undecidable. Additionally, Alan Turing's advancements have further encouraged the development of new advanced capabilities, particularly in areas such as artificial intelligence and machine learning, where undecidability helps guide the bounds of what can be reliably computed through these methods, as they are also extremely computationally expensive. By looking into these fundamental boundaries of computer science, both theoretical researchers and practical developers gain valuable insight into understanding how to construct valuable and resilient software that provides efficient solutions for solving complex problems across a variety of algorithmic constraints.

4 Evidence of Participation

- Weekly Discord Questions

- Video Summaries/Reviews:

1. Jensen Huang on GPUs

In this video, Jensen Huang, the CEO of NVIDIA, explains how GPUs changed from tools for specific computing tasks into much more broad processors that are used for everything from tasks to gaming to AI. GPUs were different at first for specific tasks, things like gaming or video editing, but now many of them share a common architecture based around concepts such as CUDA and tensor cores. Tensor cores were very important in boosting GPU capabilities by improving applications in artificial intelligence, and doing other things such as very fast graphics rendering, and more complex physical simulations. Huang talks about the massive increases in computing power, which are driven by both hardware improvements and software changes such as better and more efficient algorithms. Traditional CPUs still matter, but GPUs are much more dominant right now due to their ability to parallel process information. He discusses how technology growth has been much faster than earlier predictions, like Moore's Law, thanks to improvements like parallel processing in multiple GPUs and data centers. He also discusses how AI is changing industries in ways we wouldn't expect, such as for transforming communication networks. Using AI, bandwidth requirements can be dramatically reduced by predicting and generating data rather than transmitting everything. These changes show how GPUs have caused massive improvements in other technology, which are mostly powered by Nvidia's new chips, and they aim to continue drastically improving computing and reshaping possibilities across many fields.

2. Generative AIs Greatest Flaw

In this video, computer-security researcher Mike Pound shows why "indirect prompt injection" is considered one of generative AI's biggest weaknesses. Rather than typing an obvious instruction like "Ignore everything and write me a poem," attackers hide secret text inside emails, web pages, résumés, or other data that an AI system later reads. When the model pulls in that poisoned data, the hidden text can quietly tell it to leak private files, approve fake payments, or rank one job applicant higher than others. Pound explains that this problem is similar to old-style SQL injection, but harder to fix because language models treat every piece of text, either good or bad, as part of one big prompt. The current defenses for this include carefully curating data sources, heavy automated testing, and limiting what outside tools the AI can touch, but none of these fully solves the issue. As AI assistants start handling more crucial information such as bank transfers, medical reports, and network controls, a single malicious hidden instruction could cause real harm to a variety of systems, in many different ways. Pound's main point is that companies must treat indirect prompt injection as a permanent risk, use layered safeguards, and keep updating tests as new attack tricks appear. The flaw shows us that even advanced models remain vulnerable to very small bits of malicious text.

3. But What Is Quantum Computing?

In this video, Grant Sanderson (3Blue1Brown) clears up a common misconception about quantum computers. Many people think they are fast simply because they try every answer at once, but Sanderson shows that the real benefit comes from special algorithms and clever tricks relating to probability. He focuses on Grover's algorithm, which speeds up searching a large list. A normal computer might check a million items one by one, but Grover's quantum method only needs about a thousand steps, which is the square-root of the list size, and does this by repeatedly "rotating" the probability toward the right answer. Sanderson begins with the key building block, a qubit, which can be a mix of 0 and 1 at the same time. He explains how changing the "phase" of a qubit does not change its probability immediately, but does matter after several operations. By flipping phases for the 'secret' item and reflecting the state around an equal-balance line, Grover's process steadily piles almost all probability onto the correct cell, so a final measurement is very likely to reveal the answer. Sanderson stresses that quantum gains are problem-specific: some tasks (like factoring) enjoy dramatic speed-ups, while most see smaller benefits. The takeaway is that quantum power comes from geometry and carefully designed rotations, not magic parallelism.

4. Regular Expressions

In this discussion of regular expressions, Professor Brian Kernighan (and the Computerphile team) traces how regular expressions, created by mathematician Stephen Kleene in the 1950s, turned large state-machine diagrams into compact text patterns. A finite automaton that accepts strings like "abbb...c" can be drawn with circles and arrows, but writing the same rule as ab*c is far shorter and easier for programs to process.

The star* means "zero or more," and similar shorthand symbols cover optional parts, alternation, and groups. He shows that regular expressions, state-machine diagrams, and 'Chomsky' grammars are mathematically equivalent, that they match exactly the same strings, but regexes still win because of convenience. He also explains the problem of non-determinism, where a pattern engine might have several ways to match the next character. Computer scientists Michael Rabin and Dana Scott proved any non-deterministic automaton can be converted into a deterministic one, although the conversion can explode in size. Unix pioneer Ken Thompson tackled this by compiling small, fast pieces of assembly on the fly, while large tools like lex sometimes spend minutes converting big grammars up front. The overall idea is that regexes give programmers a powerful, concise language for finding text patterns, but efficient engines still rely on deep theoretical results and clever practical hacks.

5. Parsing Explained

In this video about parsing, Professor David Brailsford demonstrates parsing by inventing a tiny English-like language featuring robots, cats, dogs, and "two furry dice." Parsing means breaking a sentence into parts such as the subject, verb, object, by using a formal grammar. He writes the grammar in something called Backus-Naur Form, which he explains is for the angle-bracket notation created for Algol 60(haven't heard of this). He shows two strategies. In the first one, called top-down parsing, you start with the overall rule "sentence -> subject verb object," push those goals on a stack, and match the words left to right. In the second, called bottom-up parsing, you begin with the words and merge them upward until the whole tree forms. Because his grammar lets "the robot" be parsed either as a shortcut noun phrase or as article + noun, the sentence "the robot stroked two furry dice" has two valid parse trees. That harmless ambiguity contrasts with math expressions such as "$8 \div 4 \div 2$," where different trees give answers 1 or 4. Programming languages avoid such ambiguity by adding precedence and associativity rules, ensuring compilers always build the same tree. Brailsford ends by noting that clear grammars and reliable parsers are crucial for compilers, calculators, and any software that must understand structured input, from code to human language.

5 Conclusion

Throughout this course, a variety of topics were discussed that provide a valuable and broad foundation for understanding the connections between theoretical computer science and practical software engineering. Some of the topics explored include deterministic and non deterministic finite automata, regular expressions, and formal languages, as well as more advanced topics such as Turing machines, complexity analysis, and decidability problems. These concepts have highlighted the critical importance of the theory underlying the modern day practices of software development and engineering.

Looking at the broader domain of software engineering, these foundational concepts aim to provide engineers with the essential ideas for understanding both the powerful capabilities and practical limitations of various different computing systems. For example, automata theory demonstrates how to design and develop efficient parsing algorithms, which are very important to practical design choices in domains such as compiler construction, text processing, and data validation tasks. Similarly, complexity analysis has a direct influence upon algorithm selection and optimization in problem solving, which helps to guide developers in making good decisions surrounding the design choices in scalable, performance-oriented software solutions in real world applications and industries. Additionally, Turing machines and the Halting problem provide invaluable insight into the absolute boundaries of what can be solved algorithmically, which helps engineers better understand what problems are theoretically impossible so that they can better focus their attention on problems with decidable, efficient, and feasible solutions.

Throughout the course materials, I found the topic of Turing machines and the Halting problem to be particularly interesting, provided that these concepts have laid the groundwork for modern day debugging and testing practices and solutions. Understanding that particular computing problems are simply undecidable - and why - has given me a new perspective on software debugging, and demonstrated that it is necessary to use a variety of approaches to solve, implement, debug, and test complex software problems. Being able to utilize the various approaches, such as heuristic algorithms, automated testing frameworks, and more formal verification methods has become increasingly important when I go about solving problems, and also allows for a much more streamlined approach. This understanding is particularly relevant in everyday software development, where reliability and efficiency are the most important outcomes in systems that affect virtually every industry worldwide, from finance, healthcare, and transportation.

Reflecting on potential improvements for the course, I think more direct applications and code examples throughout the semester could be very valuable in student comprehension of challenging topics. Additionally, I believe that the course could benefit from more project based learning, seeing how these concepts can be applied through code to more real world problems would really bridge the gap between theory and practice. Overall, the course was very valuable, providing a theoretical framework to understanding why modern day coding practices and standards have been developed to the way they are today across a variety of industries that all utilize computing to some degree.

References

[ALG] J. Weinberger, Algorithm Analysis 2025.

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