Undecidability

What is Undecidability?

- A problem is undecidable if there is no algorithm that can solve it for all inputs (e.g., software verification)
- While it might seem we have sufficiently powerful computers to solve any problem using algorithms, there are problems, even from everyday life, that cannot be solved computationally
- A relevant example is the ATM problem: given a TM M and a string w, can we decide if M accepts input w?
- The Universal Turing Machine (UTM), proposed by Alan Turing in 1936, was designed specifically to address this problem. It works as follows:

  - TM U receives as input the string representation of TM M and string w, denoted as ⟨M,w⟩
  - U simulates M on input w
  - If M enters an accept state, U accepts

If M enters a reject state, U rejects

However, we cannot determine if this machine will halt. On some inputs, it might run forever, and no algorithm can answer this question

Important distinction: ATM is Turing-recognizable but not decidable

A TM U can recognize ATM by simulating M on w

If M accepts w, U will eventually accept

If M doesn't accept w, U might run forever

This limitation demonstrates the fundamental difference between recognition and decision

2. The Diagonalization Method

- Technique discovered by Cantor in 1873 used to prove certain sets are larger than others

- Cantor observed that two finite sets have the same size if elements of one set can be paired with elements of the other set

- This method can be extended to infinite sets

- For sets A and B and a function f from A to B:

- f is one-to-one if it never maps different elements to the same place (f(a) ≠ f(b) whenever a ≠ b)

- f is onto if it hits every element of B (for every b in B there is an a in A where f(a) = b)

- A and B are the same size if there exists a one-to-one, onto function f:A→B

- A function that is both one-to-one and onto is called a correspondence

- For example between N and 2N, the correspondence function f is f(n) = 2n, and thus we can state that N and 2N have the same size, even if it seems counterintuitive.

3. Countable Sets

- A set is countable if it's either finite or can be put into a correspondence with N (natural numbers)

- If an infinite set cannot be put into a correspondence with N then it is uncountable

4. Q is Countable

- Let Q = {m/n | m, n from N} be the set of positive rational numbers

- Though Q seems much larger than N, these sets are the same size

- Proof method:

  - Create an infinite matrix containing all positive rational numbers

  - Number i/j occurs in the ith row and jth column

  - Convert matrix to list using diagonalization

  - First diagonal: 1/1

  - Second diagonal: 2/1, 1/2

  - Third diagonal: 3/1, 2/2 (skip as equivalent to 1/1), 1/3

  - Continue this process to list all elements of Q

  - This creates a correspondence with N because we map every number in Q with a number in N (their index in the list just created)

5. R is Uncountable

- Let R be the set of real numbers

- Proof by contradiction:

  - Suppose a correspondence f exists between N and R

- Construct a number x that differs from every number in the correspondence

- Let f(1) = 3.14159..., f(2) = 55.555555..., f(3) = 1.1111..., etc.

- Construct x between 0 and 1

- Choose each digit of x to differ from the corresponding digit in f(n)

- First digit ≠ first digit of f(1)

- Second digit ≠ second digit of f(2)

- And by continuing so we have constructed x.

- This x is in R but not in the list

- Note: Never select 0 or 9 when constructing x to avoid issues with equivalent representations (0.1999... = 0.2000...)

- In this way we have shown that R cannot be put into a correspondence with N => R is uncountable

6. Some Languages are not Turing-Recognizable

- Let L be the set of all languages over alphabet Σ

- Show L is uncountable by giving a correspondence with B (set of infinite binary sequences)

- Let Σ* = {s1 = ε, s2, s3, ...}

- Each language A from L has a unique sequence in B: The ith bit of sequence is 1 if si is in A, 0 otherwise

- Example:

  - If A were the language of all strings starting with 0 over alphabet {0,1}:

  - Σ* = {ε, 0, 1, 00, 01, 10, 11, 000, 001, ...}

  - A = {0, 00, 01, 000, 001, ...}

  - Sequence = 0 1 0 1 1 0 0 1 1 ...

- Function f: L → B, where f(A) is the characteristic sequence of A, is a correspondence

- Therefore, as B is uncountable, L is uncountable

- Conclusion: In this way we proved that there are uncountably many languages but only countably many TMs, therefore some languages are not even TR, so they are non-TR. Some languages cannot be recognized by any TM

7. ATM is Undecidable

- ATM = {⟨M,w⟩| M is a TM and M accepts w}

- Proof by contradiction:
  - Assume ATM is decidable
  - Let H be a decider for ATM where:

    H(⟨M,w⟩) = accept if M accepts w

    reject if M doesn't accept w
  - Construct D using H as subroutine:

    D(⟨M⟩) = accept if M doesn't accept ⟨M⟩

    reject if M accepts ⟨M⟩
  - When we run D with its own description ⟨D⟩ as input:

    D(⟨D⟩) = accept if D doesn't accept ⟨D⟩

    reject if D accepts ⟨D⟩
  - This creates a paradox - D must do the opposite of what it does on its own input
  - Contradiction → ATM is undecidable
  - The diagonalization appears if we construct a matrix where:
    - Rows: all TMs (M1, M2, M3, ...)
    - Columns: string representations (⟨M1⟩, ⟨M2⟩, ⟨M3⟩, ...)
    - Cell [i,j]: "accept" if Mi accepts ⟨Mj⟩, "reject" if not
    - Contradiction appears in cell for D and ⟨D⟩, because we don't now if the cell will be 'accept' or 'reject'

NP-Completeness

1. Satisfiable Boolean Formula

- Variables that can take on values TRUE and FALSE are called Boolean variables
- Boolean operations: AND (∧), OR (∨), NOT (¬)
- A Boolean formula is an expression involving Boolean variables and operations

  Example: $\phi = (x \land y) \lor (x \land z)$
- A Boolean formula is satisfiable if some assignment of 0s and 1s to variables makes it evaluate to 1

  Example: The above formula is satisfiable with x=0, y=1, z=0

- Define SAT = {⟨ϕ⟩ | ϕ is a satisfiable Boolean formula}

2. Polynomial Time Reducibility

- We know about the notion of efficiently reducing one problem to another. We need to consider time complexity, so we talk about reducing one problem to another in polynomial time.

- A function f: $\Sigma^* \rightarrow \Sigma^*$ is polynomial time computable if some polynomial time Turing machine M exists that halts with just f(w) on its tape when started on any input w

- Language A is polynomial time reducible to language B (written A ≤P B) if:

- A polynomial time computable function f: $\Sigma \rightarrow \Sigma$ exists
- For every w, w ∈ A ⇔ f(w) ∈ B
- f is called the polynomial time reduction of A to B

- If A ≤P B and B is in P, then A is in P

- This means that if we have a problem A that reduces in polynomial time to another problem B, and we find a polynomial time solution for problem B, then we can use B's solution to find a polynomial time solution for A. This works because a composition of polynomials is still a polynomial, so it doesn't affect efficiency.

- Proof:

  - Let M be polynomial time algorithm deciding B

  - Let f be polynomial time reduction from A to B

  - Describe polynomial time algorithm N deciding A:

    N = "On input w:

      1. Compute f(w)

      2. Run M on f(w) and output whatever M outputs"

  - Because f is reduction from A to B → w ∈ A whenever f(w) ∈ B

  - Thus M accepts f(w) whenever w ∈ A

3. (3)CNF-Formula

- A literal is a Boolean variable or negated Boolean variable

- A clause is several literals connected with ∨s

- A Boolean formula is in conjunctive normal form (CNF) if it comprises several clauses connected with ∧s

- It is a 3CNF-formula if all clauses have exactly three literals

- Define 3SAT = {⟨ϕ⟩| ϕ is a satisfiable 3CNF-formula}

- Example of reduction: 3SAT reduces to CLIQUE

  - For a 3CNF formula, generate an undirected graph where:

    - Nodes are literals from 3SAT

    - Nodes are divided into k triplets (clauses from 3SAT)

    - There are edges between all nodes except:

      - Nodes in same triplet

      - Nodes with complementary values (x1 and ¬x1)

- Formula is satisfiable if and only if G has a k-clique. Let's prove both directions:
  - First, if ϕ is satisfiable → G has a k-clique:
    - In a satisfiable formula ϕ, each of the k clauses has at least one literal that evaluates to TRUE
    - We choose one TRUE literal from each clause (any one if multiple exist)
    - These k literals, connected by the ∧ operation, form a complete subgraph with k nodes in G
    - Complementary nodes (x and ¬x) cannot be connected, as we can't have both a variable and its negation as TRUE
    - The subgraph has exactly k nodes (one per clause), and they form a k-clique in G
  - Conversely, if G has a k-clique → ϕ is satisfiable:
    - A k-clique in G provides one node from each triplet
    - These nodes can't be from the same triplet (no edges between same-triplet nodes)
    - Each node represents a literal, and each triplet represents a clause
    - The k-clique can't contain complementary literals (no edges between x and ¬x nodes)
    - Therefore, the k-clique nodes give us a valid TRUE assignment for ϕ

4. NP-Complete

- A language B is NP-Complete if:

  a. B is in NP

  b. Every A in NP is polynomial time reducible to B

- Theoretical importance:
  - To prove P≠NP: show any NP problem needs more than polynomial time
  - To prove P=NP: find polynomial time solution for any NP-complete problem
- Practical importance:
  - Shows that it's pointless to search for polynomial-time algorithms for NP-complete problems
  - NP-completeness suggests no polynomial time solution exists
  - Helps researchers avoid wasting time searching for polynomial-time algorithms that likely don't exist
  - Guides focus toward approximation algorithms and heuristic solutions
- Theorems:
  - If B is NP-Complete and B is in P, then P=NP
  - If B is NP-Complete and B $\leq_P$ C for C in NP, then C is NP-Complete

5. Cook-Levin Theorem

- States that SAT is NP-Complete
- Discovered independently by Stephen Cook and Leonid Levin in 1970s
- Proof:
  1. Show SAT ∈ NP:
     - Nondeterministic TM can guess assignment and verify in polynomial time
  2. Show every A ∈ NP reduces to SAT:
     - Let N be NTM deciding A in time nk
     - For input w, construct formula ϕ that simulates N on w
     - Construct a maxtrix (computation table) of size nk × nk
     - ϕ = ϕcell ∧ ϕstart ∧ ϕmove ∧ ϕaccept where:

        - ϕcell is the AND of all i and j from 1 to nk ((OR of all s from C Xijs) AND (AND of all s,t from C where s≠t NOT Xijs OR NOT Xijt)) which refers to what can be found in each cell
        - ϕstart = X11# ∧ X12q0 ∧ X13w1 ∧ ... ∧ X1n+2w ∧ X1n+3_ ∧ ... ∧ X1nk#) represents the initial configuration, the starting one, of TM N
        - ϕaccept = OR for all i,j from 1 to nk of Xijq_accept refers to the fact that at least one cell in the tableau should be in an accepting state
        - ϕmove = AND for all i,j from 1 to nk where window (i,j) is legal. A window (i,j) is legal if it follows the transition function of NTM N. We can rewrite ϕmove = OR for a1,...,a6 legal window (Xij-1a1 ∧ Xija2 ∧ Xij+1a3 ∧ Xi+1j-1a4 ∧ Xi+1ja5 ∧ Xi+1j+1a6) which means that any window (a 2x3 portion of the tableau) must respect the transition function of NTM N.Key properties:
    - If we analyze the formula:
        - we have n^2k * l variables, where l depends only on the nondeterministic Turing machine N and not on n, so we can say we have O(n^2k) variables for ϕcell
        - ϕstart contains only the first line of the tableau so we can say it has complexity O(n^k)
        - ϕmove and ϕaccept refer to the entire tableau, so they also have complexity O(n^2k)
        - In conclusion, ϕ has complexity O(n^2k), which is polynomial, sufficient for our proof. Therefore, we can say we have a polynomial reduction from input w to ϕ, thus any language A from NP reduces to SAT => SAT is NP-complete.
        -

6. 3SAT is NP-Complete
   While we can use SAT to prove other problems are NP-complete, its particular form, 3SAT, is often preferred. To use 3SAT for proving NP-completeness of other problems, we first need to show that 3SAT itself is NP-complete.

- Proof:

  1. Show 3SAT ∈ NP (obvious because a nondeterministic TM can guess an assignment and verify each three-literal clause in polynomial time)

  2. Show NP-Completeness by modifying Cook-Levin proof:

    - Start with formula ϕ = ϕcell ∧ ϕstart ∧ ϕmove ∧ ϕaccept

    - Convert each part to CNF:

      - ϕcell: already in CNF (AND of clauses)

      - ϕstart: already CNF (AND of single variables)

      - ϕaccept: single clause (OR of variables)

- ϕmove: convert using distributive law (OR of ANDs → AND of ORs)

3. Convert to 3CNF:

   - For clauses < 3 literals: Replicate literals until 3

   - For clauses > 3 literals:

     $(a_1 \lor a_2 \lor a_3 \lor a_4) \rightarrow (a_1 \lor a_2 \lor z) \land (\neg z \lor a_3 \lor a_4)$

4. Key points:

   - Transformation preserves satisfiability

   - Size increases only by constant factor

   - Process takes polynomial time

In this way, we proved that 3SAT is NP-Complete.