COMPLEXITY THEORY: TIME COMPLEXITY

Introduction

Even when a problem is decidable and thus computationally solvable in principle, it may not be solvable in practice if the solution requires an inordinate amount of time or memory. It is therefore important determine the amount of the time, memory, or other resources required for solving computational problems.

A) Measuring Complexity

1. Definition

Let M be a deterministic Turing machine that halts on all inputs. The ***running time*** or ***time complexity*** of M is the function $f: \mathcal{N} \longrightarrow \mathcal{N}$, where $f(n)$ is the maximum number of steps that M uses on any input of length n. If $f(n)$ is the running time of M, we say that M runs in time $f(n)$ and that M is an $f(n)$ time Turing machine.

2. Time Complexity

Time complexity measures the amount of computational time an algorithm takes to complete as a function of the size of the input n.

- **Big-O Notation**: Describes an upper bound on the time complexity, ignoring constant factors and lower-order term.
- **Example 1**: If an algorithm runs in $O(n^2)$ time, its running time is at most proportional to n^2 for large n.
- **Example 2**: **T**he function $f(n) = 6n^3 + 2n^2 + 20n + 30$ has four terms, and the highest order term is $6n^3$. Disregarding the coefficient 6, we say that f is asymptotically at most n^3. The asymptotic notation or big-O notation for describing this relationship is $f(n) = O(n^3)$.

3. Analyzing Algorithms

To determine the time complexity of an algorithm, consider the number of basic operations (e.g., comparisons, additions) as a function of the input size n.

- **Example 1**: Linear Search
  - Input: Array of size n and a target value.
  - Algorithm: Check each element until the target is found.
  - Time Complexity: O(n) because, in the worst case, it examines each element once.
- **Example 2**: Binary Search
  - Input: Sorted array of size n and a target value.
  - Algorithm: Repeatedly divide the search interval in half.
  - Time Complexity: O(log n) because it halves the interval at each step.

B) The Class P

1. Definition

The class of decision problems (languages) that can be solved by a deterministic Turing machine in polynomial time. Formally, $P=\bigcup_{k\geq 1} TIME(n^k)$.

The class P plays a central role in our theory and is important because

a) P is invariant for all models of computation that are polynomially equivalent to the deterministic single-tape Turing machine, and

b) P roughly corresponds to the class of problems that are realistically solvable on a computer.

Item a) indicates that P is a mathematically robust class. It isn't affected by the particulars of the model of computation that we are using.

Item b) indicates that P is relevant from a practical standpoint. When a problem is in P, we have a method of solving it that runs in time for some constant k. Whether this running time is practical depends on k and on the application. Of course, a running time of n^{100} is unlikely to be of any practical use. Nevertheless, calling polynomial time the threshold of practical solvability has proven to be useful. Once a polynomial time algorithm has been found for a problem that formerly appeared to require exponential time, some key insight into it has been gained, and further reductions in its complexity usually follow, often to the point of actual practical utility.

2. **Examples**
    - **Sorting Problems**: Algorithms like Merge Sort and Quick Sort run in O(n log n) time.
    - **Shortest Path**: Dijkstra's algorithm for finding the shortest path in a graph runs in $O(V^2)$ time with an adjacency matrix representation.

3. **Significance**

Problems in P are considered efficiently solvable. Polynomial time algorithms are practical for large inputs.

C) The Class NP

1. **Definition**

The class of decision problems for which a given solution can be verified as correct in polynomial time by a deterministic Turing machine. Formally, $NP=\cup_{k\geq 1}NTIME(n^k)$, where NTIME denotes the class of problems solvable by a nondeterministic Turing machine in polynomial time. In other words, NP is the class of languages that have polynomial time verifiers.

2. **Examples**
    - **Hamiltonian Cycle**: Given a graph, determine if there exists a cycle that visits each vertex exactly once. This can be verified as follows: Given a sequence of vertices, check if it forms a Hamiltonian cycle in polynomial time.
        - **Satisfiability (SAT)**: Determine if there exists an assignment of variables that makes a boolean formula true. This can be verified as follows: Given an assignment, check if it satisfies the formula in polynomial time.
3. **Significance**

Problems in NP can be verified efficiently, but it is unknown if they can be solved efficiently.

D) NP-Completeness

1. Definition

A problem L is **NP-complete** if:

1. $L \in NP$.
2. Every problem in NP is polynomial-time reducible to L (denoted $L' \leq_p L$ for all $L' \in NP$).

2. Cook-Levin Theorem

The SAT problem was the first proven NP-complete problem. Any problem in NP can be reduced to SAT in polynomial time.

3. Significance

NP-complete problems are the hardest problems in NP. If any NP-complete problem can be solved in polynomial time, then every problem in NP can be solved in polynomial time (P=NP).

THE P VERSUS NP QUESTION

As we have been saying, NP is the class of languages that are solvable in polynomial time on a nondeterministic Turing machine, or, equivalently, it is the class of languages whereby membership in the language can be verified in polynomial time. P is the class of languages where membership can be tested in polynomial time. We summarize this information as follows, where we loosely refer to polynomial time solvable as solvable "quickly."

P the class of languages for which membership can be decided quickly.

NP the class of languages for which membership can be verified quickly.

We have presented examples of languages, such as HAMPATH and CLIQUE, that are members of NP but that are not known to be in P. The power of polynomial verifiability seems to be much greater than that of polynomial decidability. But, hard as it may be to imagine, P and NP could be equal. We are unable to prove the existence of a single language in NP that is not in P.

The question of whether P = NP is one of the greatest unsolved problems in theoretical computer science and contemporary mathematics. If these classes were equal, any polynomially verifiable problem would be polynomially decidable. Most researchers believe that the two classes are not equal because people have invested enormous effort to find polynomial time algorithms for certain problems in NP, without success. Researchers also have tried proving that the classes are unequal, but that would entail showing that no fast algorithm exists to replace brute-force search. Doing so is presently beyond scientific reach. The following figure shows the two possibilities.

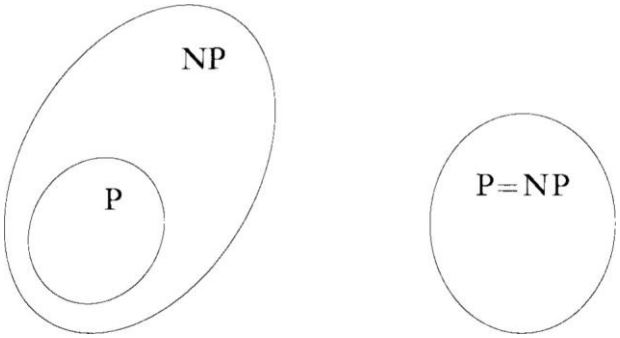

Figure 1: One of these two possibilities is correct

The best method known for solving languages in NP deterministically uses exponential time. In other words, we can prove that

$$\text{NP} \subseteq \text{EXPTIME} = \bigcup_k \text{TIME}(2^{n^k}),$$

but we don't know whether NP is contained in a smaller deterministic time complexity class.

E) NP-Complete Problems

One important advance on the P versus NP question came in the early 1970s with the work of Stephen Cook and Leonid Levin. They discovered certain problems in NP whose individual complexity is related to that of the entire class. If a polynomial time algorithm exists for any of these problems, all problems in NP would be polynomial time solvable. These problems are called ***NP-complete***. The phenomenon of NP-completeness is important for both theoretical and practical reasons.

On the theoretical side, a researcher trying to show that P is unequal to NP may focus on an NP-complete problem. If any problem in NP requires more than polynomial time, an NP-complete one does. Furthermore, a researcher attempting to prove that P equals NP only needs to find a polynomial time algorithm for an NP-complete problem to achieve this goal.

On the practical side, the phenomenon of NP-completeness may prevent wasting time searching for a nonexistent polynomial time algorithm to solve a particular problem. Even though we may not have the necessary mathematics to prove that the problem is unsolvable in polynomial time, we believe that P is unequal to NP, so proving that a problem is NP-complete is strong evidence of its nonpolynomiality.

The first NP-complete problem that we present is called the ***satisfiability problem***. Recall that variables that can take on the values TRUE and FALSE are called ***Boolean variables*** (see Section 0.2). Usually, we represent TRUE by 1 and FALSE by 0. The ***Boolean operations*** AND, OR, and NOT, represented by the symbols $\wedge$, $\vee$, and $\neg$, respectively, are described in the following list. We use the overbar as a shorthand for the $\neg$ symbol, so $\overline{x}$ means $\neg\, x$.

$$\begin{array}{lll} 0 \wedge 0 = 0 & 0 \vee 0 = 0 & \overline{0} = 1 \\ 0 \wedge 1 = 0 & 0 \vee 1 = 1 & \overline{1} = 0 \\ 1 \wedge 0 = 0 & 1 \vee 0 = 1 & \\ 1 \wedge 1 = 1 & 1 \vee 1 = 1 & \end{array}$$

A ***Boolean formula*** is an expression involving Boolean variables and operations. For example,

$$\phi = (\overline{x} \wedge y) \vee (x \wedge \overline{z})$$

is a Boolean formula. A Boolean formula is ***satisfiable*** if some assignment of 0s and 1s to the variables makes the formula evaluate to 1. The preceding formula is satisfiable because the assignment $x = 0$, $y = 1$, and $z = 0$ makes ϕ evaluate to 1. We say the assignment *satisfies* ϕ. The ***satisfiability problem*** is to test whether a Boolean formula is satisfiable. Let

$$SAT = \{\langle\phi\rangle \mid \phi \text{ is a satisfiable Boolean formula}\}.$$

Now we state the Cook–Levin theorem which links the complexity of the *SAT* problem to the complexities of all problems in NP.

Definition

A language B is ***NP-complete*** if it satisfies two conditions:

a) B is in NP, and

b) every A in NP is polynomial time reducible to B.

theorem

If B is NP-complete and B ∈ P, then P = NP.

1. **Examples**
   - **3-SAT**: A special case of SAT where each clause has exactly three literals.
     - Reduction: Any instance of SAT can be transformed into an instance of 3-SAT.
   - **Clique**: Given a graph and an integer k, determine if there is a clique of size k.
     - Reduction: The problem can be reduced from SAT or other NP-complete problems.
   - **Vertex Cover**: Given a graph and an integer k, determine if there is a set of k vertices that cover all edges.
     - Reduction: The problem can be reduced from 3-SAT.

2. **Reductions**

Reductions are used to prove NP-completeness by transforming one NP-complete problem into another in polynomial time.

Example of Reduction:

- To prove that a problem B is NP-complete, reduce a known NP-complete problem A to B.
- If $A \leq_p B$ and $B \in NP$, then B is NP-complete.

Problems

1: Analyzing Algorithm Complexity

- **Problem**: Analyze the time complexity of the following algorithm:

```
// Function to count the number of operations
int count_operations(int arr[], int n) {
    int count = 0;
    for (int i = 0; i < n; i++) {
        for (int j = i; j < n; j++) {
            count++;
        }
    }
    return count;
}
```

- **Solution**:
    - Outer loop runs n times.
    - Inner loop runs n−i times for each iteration of the outer loop.
    - Total operations:

$$\sum_{i=0}^{n-1}(n-i) = n + (n-1) + (n-2) + \ldots + 1 = \frac{n(n+1)}{2} = O(n^2)$$

Example 2: Proving NP-Completeness

- **Problem**: Show that the Clique problem is NP-complete.
- **Solution**:
    - **Verification**: Given a set of vertices, check if they form a clique in polynomial time (in NP).

- **Reduction**: Reduce 3-SAT to Clique.
  - For each clause in a 3-SAT formula, create a vertex.
  - Connect vertices with edges if their literals are not negations of each other.
  - A clique of size equal to the number of clauses exists if and only if the 3-SAT formula is satisfiable.

Example 3: Decidable vs. NP-Complete

- **Problem**: Compare a decidable problem and an NP-complete problem.
- **Solution**:
  - **Decidable Problem**: Checking if a number is prime (polynomial time).
  - **NP-Complete Problem**: Solving the Hamiltonian Cycle problem.
  - The prime-checking algorithm can always determine primality in polynomial time, while the Hamiltonian Cycle problem can be verified quickly but not necessarily solved quickly unless P=NP.