ADSA ASSIGNMENT

Naresh Kota
Roll no - A125010

Question 1

Problem. Prove that the time complexity of the recursive HEAPIFY procedure is $O(\log n)$, given the recurrence:

$$T(n) = T\left(\frac{2n}{3}\right) + O(1)$$

Solution

We are given the recurrence:

$$T(n) = T\left(\frac{2n}{3}\right) + O(1)$$

Let the constant amount of work outside the recursive call be $c > 0$. Then the recurrence can be rewritten as:

$$T(n) = T\left(\frac{2n}{3}\right) + c$$

Step 1: Expand the recurrence

Applying the recurrence repeatedly:

$$T(n) = T\left(\frac{2n}{3}\right) + c$$

$$= T\left(\left(\frac{2}{3}\right)^2 n\right) + 2c$$

$$= T\left(\left(\frac{2}{3}\right)^3 n\right) + 3c$$

Continuing in this manner, after k steps we obtain:

$$T(n) = T\left(\left(\frac{2}{3}\right)^k n\right) + kc$$

Step 2: Determine when recursion stops

The recursion terminates when the problem size becomes constant:

$$\left(\frac{2}{3}\right)^k n \leq 1$$

Taking natural logarithms:

$$k \ln\left(\frac{2}{3}\right) \leq -\ln n$$

Since $\ln(2/3) < 0$, dividing reverses the inequality:

$$k \geq \frac{\ln n}{\ln(3/2)}$$

Thus,

$$k = \Theta(\log n)$$

Step 3: Substitute back

At termination:

$$T\left(\left(\frac{2}{3}\right)^k n\right) = T(1) = O(1)$$

Hence,

$$T(n) = O(1) + kc = O(\log n)$$

Conclusion

$$\boxed{T(n) = O(\log n)}$$

Question 2

Problem. In an array of size n representing a binary heap (using 1-based indexing), prove that all leaf nodes are located at indices

$$\left\lfloor \frac{n}{2} \right\rfloor + 1 \text{ to } n.$$

Solution

A binary heap is a **complete binary tree** that is stored in an array in level-order form.

Array Representation of a Binary Heap

Let the heap be stored in an array $A[1 \dots n]$. The structural properties of a binary heap imply:

- The parent of a node at index i is at index:

$$\left\lfloor \frac{i}{2} \right\rfloor$$

- The left child of a node at index i is at index:

$$2i$$

- The right child of a node at index i is at index:

$$2i + 1$$

Definition of a Leaf Node

A node in a binary tree is called a **leaf node** if it has no children.

Thus, a node at index i is a leaf if:

$$2i > n \quad \text{and} \quad 2i + 1 > n$$

Step 1: Consider indices greater than $\lfloor \frac{n}{2} \rfloor$

Let:

$$i > \left\lfloor \frac{n}{2} \right\rfloor$$

Then:

$$2i > n$$

Since the left child index itself exceeds n, the right child index $2i + 1$ also exceeds n. Hence, node i has no children and is therefore a **leaf node**.

Step 2: Consider indices less than or equal to $\lfloor \frac{n}{2} \rfloor$

Let:

$$i \leq \left\lfloor \frac{n}{2} \right\rfloor$$

Then:

$$2i \leq n$$

Thus, node i has at least one child and cannot be a leaf node. Such nodes are called **internal nodes**.

Step 3: Classification of nodes

From the above analysis:

- Indices 1 to $\lfloor \frac{n}{2} \rfloor$ correspond to internal nodes
- Indices $\lfloor \frac{n}{2} \rfloor + 1$ to n correspond to leaf nodes

Conclusion

Therefore, all leaf nodes in an n-element binary heap are located at indices:

$$\boxed{\left\lfloor \frac{n}{2} \right\rfloor + 1 \text{ to } n}$$

Question 3

(a) Number of nodes at height h in a binary heap

Problem. Prove that in an n-element binary heap, the number of nodes at height h is at most:

$$\left\lfloor \frac{n}{2^{h+1}} \right\rfloor$$

Solution

Definition of Height

The **height** of a node in a binary tree is defined as the number of edges on the longest downward path from that node to a leaf.

Thus:

- A leaf node has height 0
- A node whose children are leaves has height 1

Observation

A node of height h must have a subtree of height h rooted at that node.

Step 1: Minimum number of nodes in a subtree of height h

The smallest complete binary tree of height h contains:

$$1 + 2 + 4 + \cdots + 2^h$$

This is a geometric series whose sum is:

$$2^{h+1} - 1$$

Thus, any node of height h must dominate at least:

$$2^{h+1} - 1$$

nodes in the heap.

Step 2: Bounding the number of nodes at height h

Suppose there are k nodes of height h in the heap.

Then the total number of nodes in the heap must satisfy:

$$n \geq k(2^{h+1} - 1)$$

Rearranging:

$$k \leq \frac{n}{2^{h+1} - 1}$$

Since:

$$2^{h+1} - 1 > 2^h$$

we obtain:

$$k < \frac{n}{2^{h+1}}$$

Conclusion

Therefore, the number of nodes at height h is at most:

$$\boxed{\left\lfloor \frac{n}{2^{h+1}} \right\rfloor}$$

(b) Time complexity of the Build-Heap algorithm

Problem. Using the result of part (a), prove that the BUILD-HEAP algorithm runs in linear time.

Solution

Overview of the Build-Heap Algorithm

The BUILD-HEAP algorithm constructs a heap from an unordered array by calling HEAPIFY on all internal nodes, starting from the lowest level and moving upward.

Cost of Heapify

The running time of HEAPIFY on a node is proportional to the height of that node.

If a node has height h, then:

$$\text{Cost of Heapify} = O(h)$$

Step 1: Group nodes by height

From part (a), the number of nodes of height h is at most:

$$\frac{n}{2^{h+1}}$$

Step 2: Total cost computation

The total running time of Build-Heap is the sum of the costs of heapifying all nodes:

$$T(n) = \sum_{h=0}^{\lfloor \log n \rfloor} \left(\frac{n}{2^{h+1}} \cdot O(h) \right)$$

Factoring out $O(n)$:

$$T(n) = O(n) \sum_{h=0}^{\lfloor \log n \rfloor} \frac{h}{2^h}$$

Step 3: Convergence of the series

The series:

$$\sum_{h=0}^{\infty} \frac{h}{2^h}$$

is a convergent series and evaluates to a constant.

Hence:

$$\sum_{h=0}^{\lfloor \log n \rfloor} \frac{h}{2^h} = O(1)$$

Final Calculation

$$T(n) = O(n) \cdot O(1) = O(n)$$

Conclusion

$$\boxed{\text{Build-Heap runs in linear time } O(n)}$$

This result is non-trivial and highlights the efficiency of the bottom-up heap construction algorithm.

Question 4

Problem. Explain LU decomposition of a matrix using Gaussian Elimination. Describe the method in detail and explain how it is used to solve a system of linear equations.

Solution

Introduction

LU decomposition is a matrix factorization technique in which a given square matrix A is expressed as the product of two triangular matrices:

$$A = LU$$

where:

- L is a **lower triangular matrix** with unit diagonal entries
- U is an **upper triangular matrix**

This decomposition is widely used in numerical methods and algorithms for efficiently solving systems of linear equations.

Prerequisite

LU decomposition exists without row pivoting if all leading principal minors of A are non-zero.

Step 1: Gaussian Elimination

Consider a system of linear equations:

$$Ax = b$$

where $A \in \mathbb{R}^{n \times n}$.

Gaussian elimination transforms matrix A into an upper triangular matrix U by eliminating entries below the main diagonal.

At the k-th step $(1 \leq k \leq n-1)$, the entries a_{ik} for $i > k$ are eliminated using the multiplier:

$$m_{ik} = \frac{a_{ik}}{a_{kk}}$$

The corresponding row operation is:

$$R_i \leftarrow R_i - m_{ik} R_k$$

After completing all elimination steps, the matrix becomes upper triangular, denoted by U.

Step 2: Construction of the Lower Triangular Matrix L

The multipliers m_{ik} used during Gaussian elimination are stored in the matrix L.

The structure of L is:

$$L = \begin{bmatrix} 1 & 0 & 0 & \cdots & 0 \\ m_{21} & 1 & 0 & \cdots & 0 \\ m_{31} & m_{32} & 1 & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & 0 \\ m_{n1} & m_{n2} & m_{n3} & \cdots & 1 \end{bmatrix}$$

The diagonal entries of L are set to 1 because no scaling of pivot rows is performed.

Step 3: Verification of LU Decomposition

Each elimination step corresponds to multiplying A by an elementary lower triangular matrix.

Combining all elimination steps gives:

$$A = LU$$

Thus, Gaussian elimination implicitly computes the LU decomposition of A.

Step 4: Solving a Linear System Using LU Decomposition

Given:

$$Ax = b$$

and $A = LU$, we solve the system in two stages:

(i) Forward Substitution Solve:

$$Ly = b$$

This is done in $O(n^2)$ time since L is lower triangular.

(ii) Backward Substitution Solve:

$$Ux = y$$

This is also done in $O(n^2)$ time since U is upper triangular.

Computational Complexity

- LU decomposition: $O(n^3)$
- Forward substitution: $O(n^2)$
- Backward substitution: $O(n^2)$

Once LU is computed, multiple systems with different right-hand sides can be solved efficiently.

Conclusion

LU decomposition converts a complex system of linear equations into two simpler triangular systems. It improves computational efficiency and numerical stability, making it a fundamental technique in numerical linear algebra and algorithm design.

$$\boxed{A = LU \text{ is an efficient factorization for solving linear systems}}$$

Question 5

Problem. Solve the following recurrence relation arising in the LUP decomposition solve procedure and determine its asymptotic time complexity:

$$T(n) = \sum_{i=1}^{n} \left[O(1) + \sum_{j=1}^{i-1} O(1) \right] + \sum_{i=1}^{n} \left[O(1) + \sum_{j=i+1}^{n} O(1) \right]$$

Solution

The given recurrence represents the total work done by two nested summation processes. We analyze each component separately.

Step 1: Simplification of the inner summations

Consider the first inner summation:

$$\sum_{j=1}^{i-1} O(1)$$

Since each term contributes a constant amount of work:

$$\sum_{j=1}^{i-1} O(1) = O(i-1) = O(i)$$

Now consider the second inner summation:

$$\sum_{j=i+1}^{n} O(1)$$

The number of terms is $(n-i)$, hence:

$$\sum_{j=i+1}^{n} O(1) = O(n-i)$$

Step 2: Substitute simplified expressions

Substituting the simplified results into the original expression:

$$T(n) = \sum_{i=1}^{n} [O(1) + O(i)] + \sum_{i=1}^{n} [O(1) + O(n-i)]$$

Dropping constant terms:

$$T(n) = \sum_{i=1}^{n} O(i) + \sum_{i=1}^{n} O(n-i)$$

Step 3: Evaluate each summation

We evaluate the two summations separately.

First Summation

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

Second Summation

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

Step 4: Combine the results

Adding both contributions:

$$T(n) = O(n^2) + O(n^2)$$

$$T(n) = O(n^2)$$

Conclusion

The total time complexity of the given recurrence relation is:

$$\boxed{T(n) = O(n^2)}$$

This result is consistent with the computational complexity of the forward and backward substitution steps in LUP decomposition.

Question 6

Problem. Prove that if a matrix A is non-singular, then its Schur complement is also non-singular.

Solution

Introduction

The Schur complement is an important concept in matrix theory and numerical linear algebra. It plays a crucial role in block matrix factorization, LU decomposition, and stability analysis of algorithms.

Matrix Partitioning

Let A be a square matrix partitioned as:

$$A = \begin{bmatrix} B & C \\ D & E \end{bmatrix}$$

where:

- B is a square submatrix of A
- B is assumed to be non-singular (invertible)

Definition: Schur Complement

The **Schur complement** of block B in matrix A is defined as:

$$S = E - DB^{-1}C$$

Step 1: Block Matrix Factorization

Using block Gaussian elimination, matrix A can be factorized as:

$$A = \begin{bmatrix} I & 0 \\ DB^{-1} & I \end{bmatrix} \begin{bmatrix} B & C \\ 0 & S \end{bmatrix}$$

Both matrices on the right-hand side are block triangular matrices.

Step 2: Determinant of the Block Factors

The determinant of a triangular block matrix is the product of the determinants of its diagonal blocks.

Hence:

$$\det(A) = \det(B) \cdot \det(S)$$

Step 3: Use Non-Singularity of A

Since matrix A is non-singular, by definition:

$$\det(A) \neq 0$$

Also, since B is invertible:

$$\det(B) \neq 0$$

Substituting into the determinant equation:

$$\det(B) \cdot \det(S) \neq 0$$

This implies:

$$\det(S) \neq 0$$

Step 4: Interpretation

A non-zero determinant implies that matrix S is invertible, i.e., non-singular.

Conclusion

Therefore, we conclude that:

$$\boxed{\text{If } A \text{ is non-singular and } B \text{ is invertible, then the Schur complement } S \text{ is non-singular}}$$

Question 7

Problem. Explain why positive-definite matrices are suitable for LU decomposition using the recursive strategy and why pivoting is not required in this case.

Solution

Introduction

LU decomposition using a recursive or Gaussian elimination strategy may fail if a pivot element becomes zero. To avoid this, pivoting (row exchanges) is often used. However, for **positive-definite matrices**, LU decomposition can be performed safely *without pivoting*. We justify this formally below.

Definition: Positive-Definite Matrix

A real symmetric matrix $A \in \mathbb{R}^{n\times n}$ is called **positive-definite** if:

$$x^T A x > 0 \quad \forall x \in \mathbb{R}^n,\ x \neq 0$$

Key Property: Leading Principal Minors

A fundamental theorem in linear algebra states that:

> A symmetric matrix A is positive-definite if and only if all its leading principal minors are positive.

That is,

$$\det(A_k) > 0 \quad \text{for } k = 1, 2, \ldots, n$$

where A_k denotes the $k \times k$ leading principal submatrix of A.

Step 1: Pivots in LU Decomposition

In LU decomposition without pivoting:

- The pivot at step k is the diagonal element u_{kk} of the upper triangular matrix U
- A zero pivot would make division impossible

For LU decomposition, the pivot satisfies:

$$u_{kk} = \frac{\det(A_k)}{\det(A_{k-1})}$$

with the convention $\det(A_0) = 1$.

Step 2: Positivity of Pivots

Since A is positive-definite:

$$\det(A_k) > 0 \quad \text{and} \quad \det(A_{k-1}) > 0$$

Therefore:

$$u_{kk} > 0 \quad \forall k$$

Thus, **no pivot is ever zero or negative**.

Step 3: Consequences for Recursive LU Decomposition

Because all pivots are strictly positive:

- Division by zero never occurs
- Recursive elimination steps are well-defined
- Numerical stability is improved

Hence, **pivoting is unnecessary**.

Step 4: Algorithmic Significance

This property is particularly important in:

- Cholesky decomposition (a specialized LU decomposition)
- Efficient numerical solvers
- Recursive matrix algorithms

Conclusion

We conclude that:

$$\boxed{\text{Positive-definite matrices always admit LU decomposition without pivoting}}$$

This guarantees correctness and stability of the recursive LU strategy.

Question 8

Problem. While finding an augmenting path in a graph, should Breadth First Search (BFS) or Depth First Search (DFS) be used? Justify your answer with proper reasoning.

Solution

Introduction

Augmenting paths play a central role in matching algorithms for graphs. They are used to increase the size of a matching by alternating between unmatched and matched edges. The choice of graph traversal method significantly affects the efficiency of the algorithm.

Definition: Matching

A **matching** M in a graph $G = (V, E)$ is a set of edges such that no two edges share a common vertex.

A vertex is called:

- **Matched** if it is incident to an edge in M
- **Free** (or unmatched) otherwise

Definition: Augmenting Path

An **augmenting path** with respect to a matching M is a simple path that:

- Starts and ends at free vertices
- Alternates between edges not in M and edges in M

Augmenting along such a path increases the size of the matching by exactly one.

Step 1: Role of Graph Traversal

To find an augmenting path, the graph must be explored from free vertices. Two natural choices are:

- Depth First Search (DFS)
- Breadth First Search (BFS)

Step 2: Limitations of DFS

DFS explores one path deeply before considering alternatives. As a result:

- DFS may find a very long augmenting path
- It does not guarantee the shortest augmenting path
- This can lead to a large number of augmentation steps

Hence, DFS may result in poor worst-case performance.

Step 3: Advantages of BFS

BFS explores vertices level by level. Therefore:

- BFS always finds the shortest augmenting path (minimum number of edges)
- Shorter augmenting paths lead to faster convergence
- The total number of augmentations is reduced

Step 4: Algorithmic Justification

Efficient matching algorithms such as the Hopcroft–Karp algorithm explicitly use BFS to:

- Construct layered graphs
- Identify multiple shortest augmenting paths in one phase
- Achieve improved time complexity

This demonstrates the theoretical and practical superiority of BFS in this context.

Conclusion

Based on the above analysis, we conclude that:

$$\boxed{\text{Breadth First Search (BFS) should be used to find augmenting paths}}$$

BFS ensures correctness, efficiency, and optimal performance in matching algorithms.

Question 9

Problem. Explain in detail why Dijkstra's algorithm cannot be applied to graphs containing negative edge weights.

Solution

Introduction

Dijkstra's algorithm is a greedy algorithm used to compute the single-source shortest paths in a weighted graph. Its correctness depends on a fundamental assumption regarding edge weights.

Key Assumption of Dijkstra's Algorithm

Dijkstra's algorithm assumes that:

> Once a vertex is extracted as the minimum-distance vertex from the priority queue, its shortest path distance is final and will never be improved.

This assumption holds **only if all edge weights are non-negative**.

Step 1: Greedy Selection Mechanism

At each iteration:

- The vertex u with the smallest tentative distance is selected
- The algorithm then relaxes all outgoing edges from u
- Vertex u is marked as finalized

After finalization, the algorithm never revisits u.

Step 2: Effect of Negative Edge Weights

If the graph contains a negative edge weight:

- A shorter path to an already finalized vertex may exist via another vertex
- This shorter path can only be discovered later

However, since Dijkstra's algorithm does not allow reprocessing of finalized vertices, it fails to correct this shorter distance.

Step 3: Illustrative Explanation

Suppose a vertex u is finalized with distance $d(u)$. If there exists a path:

$$s \to v \to u$$

such that:

$$d(s, v) + w(v, u) < d(u)$$

where $w(v, u) < 0$, then the algorithm has already made an incorrect decision.

Step 4: Consequences

As a result:

- The computed distances may not be shortest paths
- The algorithm produces incorrect results
- The greedy strategy breaks down

Conclusion

Therefore, we conclude that:

$$\boxed{\text{Dijkstra's algorithm cannot handle graphs with negative edge weights}}$$

In such cases, algorithms like BELLMAN–FORD must be used.

Question 10

Problem. Prove that every connected component of the symmetric difference of two matchings in a graph is either a path or an even-length cycle.

Solution

Introduction

The concept of symmetric difference of matchings is fundamental in matching theory and is widely used in the analysis of augmenting paths and matching algorithms.

Definition: Matching

A **matching** in a graph $G = (V, E)$ is a set of edges such that no two edges share a common vertex.

Let M_1 and M_2 be two matchings in G.

Definition: Symmetric Difference

The **symmetric difference** of M_1 and M_2 is defined as:

$$M_1 \oplus M_2 = (M_1 \setminus M_2) \cup (M_2 \setminus M_1)$$

It consists of edges that belong to exactly one of the two matchings.

Step 1: Degree of vertices in $M_1 \oplus M_2$

Since M_1 and M_2 are matchings:

- Each vertex is incident to at most one edge in M_1
- Each vertex is incident to at most one edge in M_2

Therefore, in the graph formed by $M_1 \oplus M_2$, the degree of any vertex is at most:

$$\deg(v) \leq 2$$

Step 2: Structure of graphs with maximum degree 2

A graph in which every vertex has degree at most 2 can only consist of:

- Isolated vertices
- Simple paths
- Simple cycles

Isolated vertices correspond to vertices not incident to any edge in $M_1 \oplus M_2$.

Step 3: Alternating structure of edges

In any connected component containing edges:

- Edges must alternate between M_1 and M_2
- No two consecutive edges can belong to the same matching

Otherwise, a vertex would be incident to two edges from the same matching, contradicting the definition of a matching.

Step 4: Length of cycles

In a cycle:

- The edges alternate between M_1 and M_2
- Equal number of edges must come from each matching

Hence, the total number of edges in the cycle must be even.

Conclusion

We conclude that every connected component of $M_1 \oplus M_2$ is either:

- A simple path, or
- An even-length cycle

$$\boxed{\text{Each connected component of } M_1 \oplus M_2 \text{ is a path or an even-length cycle}}$$

Question 11

Problem. Define the complexity class **Co-NP**. Explain its meaning, properties, and significance with suitable examples.

Solution

Introduction

In computational complexity theory, decision problems are classified based on the resources required to solve or verify them. The class **Co-NP** is one of the fundamental complexity classes closely related to **NP**.

Formal Definition of Co-NP

A language (decision problem) L belongs to the class **Co-NP** if and only if its complement $\overline{L}$ belongs to NP.

Formally:

$$\text{Co-NP} = \{\, L \mid \overline{L} \in \text{NP} \,\}$$

Here, the complement language $\overline{L}$ is defined as:

$$\overline{L} = \{x \mid x \notin L\}$$

Interpretation of the Definition

The definition implies that:

- Problems in NP have efficiently verifiable **YES** instances
- Problems in Co-NP have efficiently verifiable **NO** instances

Thus, for a problem $L \in \text{Co-NP}$:

- If the correct answer is **NO**, there exists a certificate
- This certificate can be verified in polynomial time
- No such guarantee is required for YES instances

Certificate-Based Verification

Let x be an input instance. If $x \notin L$ (i.e., the answer is NO), then there exists a certificate c such that:

$$V(x, c) = \text{TRUE}$$

where V is a polynomial-time verification algorithm.

Relationship Between NP and Co-NP

- NP focuses on efficient verification of YES answers
- Co-NP focuses on efficient verification of NO answers
- It is an open problem whether:

$$\text{NP} = \text{Co-NP}$$

Most complexity theorists believe that NP $\neq$ Co-NP.

Examples of Co-NP Problems

Example 1: UNSAT

- **SAT**: Is there an assignment that satisfies a Boolean formula? (NP)
- **UNSAT**: Is there no assignment that satisfies the formula? (Co-NP)

For UNSAT, a certificate can be a proof showing that all possible assignments fail.

Example 2: TAUTOLOGY Given a Boolean formula, determine whether it evaluates to TRUE for all assignments. This problem is in Co-NP.

Significance of Co-NP

The class Co-NP is important in:

- Proof complexity
- Program verification
- Cryptography and security assumptions
- Understanding the limits of efficient computation

Many problems involving universal guarantees naturally belong to Co-NP.

Conclusion

We conclude that:

$$\boxed{\text{Co-NP is the class of decision problems whose NO instances can be verified in polynomial time}}$$

This class plays a central role in theoretical computer science and complexity theory.

Question 12

Problem. Given a Boolean circuit whose output is claimed to be TRUE, explain in detail how the correctness of this result can be verified in polynomial time using Depth First Search (DFS).

Solution

Introduction

The Boolean Circuit Value Problem (BCVP) asks whether the output of a given Boolean circuit evaluates to TRUE for a specified input assignment. Although computing the output may appear complex, verifying a claimed TRUE output can be done efficiently. This establishes the problem as a member of the class NP.

Representation of a Boolean Circuit

A Boolean circuit can be represented as a **directed acyclic graph (DAG)**:

- Each vertex represents a logic gate (AND, OR, NOT, etc.)
- Directed edges represent the flow of signals between gates
- Input nodes correspond to Boolean variables or constants (0 or 1)
- The circuit has a unique output gate

Because the circuit is acyclic, no feedback loops exist.

Objective of Verification

Given:

- A Boolean circuit C
- A specific input assignment
- A claim that the output of C is TRUE

The goal is to verify the correctness of this claim efficiently, without recomputing the circuit in an exponential manner.

Step 1: Initiating DFS from the Output Gate

Verification begins by performing a Depth First Search (DFS) starting from the output gate of the circuit.

DFS ensures that:

- All gates contributing to the output are visited
- No irrelevant gates are evaluated

Step 2: Recursive Evaluation of Gates

For each gate visited during DFS:

- Recursively evaluate the values of its input gates
- Apply the Boolean operation associated with the gate
- Store the computed result to avoid redundant evaluations

Since the circuit is a DAG, each gate is evaluated exactly once.

Step 3: Handling Base Cases

The DFS recursion terminates at input gates:

- Variable nodes return the value specified by the input assignment
- Constant nodes return their fixed Boolean values

Step 4: Time Complexity Analysis

Let:

- $|V|$ = number of gates
- $|E|$ = number of wires (connections)

DFS traversal takes:

$$O(|V| + |E|)$$

Each gate evaluation requires constant time.
Hence, the total verification time is:

$$O(|V| + |E|)$$

This is polynomial in the size of the circuit.

Step 5: Correctness Argument

If the DFS-based evaluation produces TRUE at the output gate, then the claimed output is correct. If it produces FALSE, the claim is invalid.

Thus, the verification procedure is both:

- Correct
- Efficient

Conclusion

We conclude that:

$$\boxed{\text{The correctness of a TRUE Boolean circuit output can be verified in polynomial time using DFS}}$$

This demonstrates that the Boolean Circuit Value Problem belongs to the class NP.

Question 13

Problem. Prove that the **3-SAT** problem is **NP-Hard**. Also explain its membership in NP.

Solution

Introduction

The Boolean satisfiability problem (SAT) is the first problem proven to be NP-Complete. The 3-SAT problem is a restricted version of SAT in which each clause contains exactly three literals. Despite this restriction, 3-SAT remains computationally difficult. We prove this by showing that 3-SAT is NP-Hard and belongs to NP.

Definition: 3-SAT

An instance of 3-SAT consists of a Boolean formula ϕ in *conjunctive normal form (CNF)* such that:

- $\phi = C_1 \wedge C_2 \wedge \cdots \wedge C_m$
- Each clause C_i contains exactly three literals
- A literal is either a variable x or its negation $\neg x$

The question is whether there exists a truth assignment to the variables that satisfies all clauses.

Step 1: 3-SAT is in NP

To show that 3-SAT $\in$ NP, we demonstrate polynomial-time verification.

- Given a candidate truth assignment to all variables
- Evaluate each clause by checking its three literals
- Each clause evaluation takes constant time
- All clauses can be checked in $O(m)$ time

Since m is polynomial in the input size, verification is polynomial.
Thus:

$$\text{3-SAT} \in \text{NP}$$

Step 2: SAT is NP-Complete

It is a well-established result (Cook–Levin Theorem) that:

$$\text{SAT is NP-Complete}$$

This means:

- SAT $\in$ NP
- Every problem in NP can be reduced to SAT in polynomial time

Step 3: Polynomial-Time Reduction from SAT to 3-SAT

To prove NP-Hardness of 3-SAT, we show:

$$\text{SAT} \leq_p \text{3-SAT}$$

Reduction Idea Given an arbitrary CNF formula (with clauses of any length), transform it into an equivalent 3-CNF formula by:

- Breaking long clauses into multiple clauses of length 3
- Introducing new auxiliary variables

Example A clause with more than three literals:

$$(x_1 \lor x_2 \lor x_3 \lor x_4)$$

is transformed into:

$$(x_1 \lor x_2 \lor y_1) \land (\neg y_1 \lor x_3 \lor x_4)$$

where y_1 is a new variable.

This transformation:

- Preserves satisfiability
- Increases formula size only linearly
- Runs in polynomial time

Step 4: NP-Hardness Argument

Since:

- SAT is NP-Complete
- SAT reduces to 3-SAT in polynomial time

It follows that:

$$\text{3-SAT is NP-Hard}$$

Conclusion

Combining the above results:

- 3-SAT $\in$ NP
- 3-SAT is NP-Hard

Therefore:

$$\boxed{\text{3-SAT is NP-Complete (and hence NP-Hard)}}$$

Question 14

Problem. Discuss whether the **2-SAT** problem is NP-Hard. Explain in detail how the problem can be solved in polynomial time.

Solution

Introduction

The 2-SAT problem is a special case of the Boolean satisfiability problem in which each clause contains exactly two literals. Unlike 3-SAT, which is NP-Complete, 2-SAT admits an efficient polynomial-time solution. We explain both its algorithmic solution and its complexity classification.

Definition: 2-SAT

An instance of 2-SAT consists of:

- A Boolean formula in conjunctive normal form (CNF)
- Each clause has the form $(a \lor b)$, where a and b are literals

The objective is to determine whether there exists a truth assignment satisfying all clauses.

Step 1: Conversion to Implication Form

Each clause $(a \lor b)$ is logically equivalent to:

$$(\neg a \Rightarrow b) \quad \text{and} \quad (\neg b \Rightarrow a)$$

Thus, every clause can be replaced by two implications.

Step 2: Construction of the Implication Graph

Using the implications:

- Create a directed graph G
- Each literal is represented as a vertex
- Each implication corresponds to a directed edge

This graph is known as the **implication graph**.

Step 3: Strongly Connected Components (SCC)

A fundamental theorem for 2-SAT states:

> A 2-SAT instance is satisfiable if and only if no variable x and its negation $\neg x$ belong to the same strongly connected component of the implication graph.

Step 4: Algorithmic Solution

- Compute SCCs of the implication graph using:
  - Kosaraju's algorithm, or
  - Tarjan's algorithm
- For each variable x, check whether x and $\neg x$ lie in the same SCC
- If they do, the formula is unsatisfiable
- Otherwise, the formula is satisfiable

—

Step 5: Time Complexity Analysis

Let:

- V = number of literals
- E = number of implications

Both SCC algorithms run in:

$$O(V + E)$$

Hence, 2-SAT is solvable in linear time.

Step 6: Complexity Classification

Since 2-SAT has a polynomial-time algorithm:

$$\text{2-SAT} \in \text{P}$$

Therefore:

$$\text{2-SAT is not NP-Hard (unless P = NP)}$$

Conclusion

We conclude that:

$$\boxed{\text{2-SAT is solvable in polynomial time and is not NP-Hard}}$$
