Advanced Data Structures and Algorithms

Comprehensive Assignment Solutions

Arya Raag
Student ID: A125002
M.Tech (Computer Science and Engineering)

January 5, 2026

Contents

1 Heap Operations and Analysis

Question 1. Prove that the time complexity of the recursive Heapify operation is $O(\log n)$ using the recurrence relation:

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

Detailed Solution:

Problem Analysis

The `Max-Heapify` operation is a fundamental procedure used to maintain the max-heap property. When called on a node i, it assumes that the binary trees rooted at Left(i) and Right(i) are max-heaps, but $A[i]$ might be smaller than its children. The algorithm "floats" the value at $A[i]$ down the tree.

The worst-case running time occurs when the bottom level of the tree is exactly half full. In this specific configuration, the left subtree contains more nodes than the right subtree. Specifically, the left subtree can have size at most $\frac{2n}{3}$.

Formal Proof via Recurrence Solving

The recurrence relation governing the worst-case time complexity is given by:

$$T(n) \leq T\left(\frac{2n}{3}\right) + \Theta(1)$$

Here, $\Theta(1)$ represents the constant time required to compare the node with its children and perform a swap if necessary.

To solve this, we can employ the expansion method (also known as the iteration method).

Step 1: Expand the Recurrence

We iteratively substitute the recurrence into itself:

$$\begin{aligned}
T(n) &= T\left(\frac{2}{3}n\right) + c \\
&= \left[T\left(\frac{2}{3}\cdot\frac{2}{3}n\right) + c\right] + c = T\left(\left(\frac{2}{3}\right)^2 n\right) + 2c \\
&= T\left(\left(\frac{2}{3}\right)^3 n\right) + 3c \\
&\vdots \\
&= T\left(\left(\frac{2}{3}\right)^k n\right) + k\cdot c
\end{aligned}$$

Step 2: Determine the Stopping Condition
The recursion terminates when the problem size reduces to a base case, typically $n = 1$. Let k be the depth of recursion where this occurs:

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

Solving for k:

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

Taking the logarithm (base 3/2) on both sides:

$$k = \log_{3/2} n$$

Using the change of base formula, we know that $\log_{3/2} n = \frac{\ln n}{\ln 1.5}$. Since $\ln 1.5$ is a constant, $k \approx c' \ln n$. Thus, $k = O(\log n)$.

Step 3: Calculate Total Cost
Substituting k back into the expanded equation:

$$T(n) = T(1) + O(1)\cdot \log_{3/2} n$$

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

Conclusion

The height of a binary heap is logarithmic with respect to the number of nodes. Since `Heapify` performs constant work at each level of the tree and descends at most the height of the tree, the time complexity is:

$$T(n) = O(\log n)$$

Question 2. In an array of size n representing a binary heap, prove that all leaf nodes are located at indices from $\lfloor \frac{n}{2} \rfloor + 1$ to n.

Detailed Solution:

Properties of Array-Based Heaps

A binary heap is typically implemented using an array $A[1 \dots n]$. For any node located at index i:

- The index of the **Parent** is $\lfloor i/2 \rfloor$.
- The index of the **Left Child** is $2i$.
- The index of the **Right Child** is $2i + 1$.

Definition of a Leaf Node

A node is defined as a *leaf* if it has no children. In the context of the array representation, a node at index i is a leaf if and only if its left child index exceeds the bounds of the array. That is, the node i has no children if:

$$\text{Left}(i) > n$$

Substituting the formula for the left child:

$$2i > n$$

Derivation of the Index Range

We solve the inequality $2i > n$ for i:

$$i > \frac{n}{2}$$

Since array indices must be integers, the smallest integer i that satisfies strictly $i > n/2$ is:

$$i_{\min} = \left\lfloor \frac{n}{2} \right\rfloor + 1$$

Conversely, the largest possible index in an array of size n is simply n. Therefore, any node with an index i such that:

$$\left\lfloor \frac{n}{2} \right\rfloor + 1 \leq i \leq n$$

does not have a left child (and consequently, cannot have a right child, as heaps are filled from left to right).

Conclusion

We have rigorously shown that the property of being a leaf node corresponds strictly to the index condition $2i > n$. Thus, the leaves of the heap occupy precisely the second half of the array, specifically the range:

$$\left[\left\lfloor \frac{n}{2} \right\rfloor + 1, \ldots, n\right]$$

Question 3. (a) Show that in any heap containing n elements, the number of nodes at height h is at most $\lceil \frac{n}{2^{h+1}} \rceil$.
(b) Using the above result, prove that the time complexity of the Build-Heap algorithm is $O(n)$.

Detailed Solution:

Part (a): Nodes at Height h

Let h be the height of a node, defined as the number of edges on the longest simple path from the node down to a leaf. Leaves are at height 0.

By induction, we can see the structure of a complete binary tree:

- At height 0 (leaves), we have approximately $n/2$ nodes.
- At height 1, we have approximately $n/4$ nodes.
- Generally, at height h, we have roughly $n/2^{h+1}$ nodes.

Formally, the number of nodes of height h in any n-element heap is bounded by:

$$N_h \leq \left\lceil \frac{n}{2^{h+1}} \right\rceil$$

Part (b): Build-Heap Complexity Analysis

The `Build-Max-Heap` algorithm works by calling `Max-Heapify` on all non-leaf nodes, starting from the last non-leaf node down to the root.

The cost of `Max-Heapify` on a node of height h is $O(h)$. Therefore, the total cost $T(n)$ is the sum of the costs for all nodes:

$$T(n) = \sum_{h=0}^{\lfloor \lg n \rfloor} (\text{number of nodes at height } h) \cdot O(h)$$

Substituting the bound from Part (a):

$$T(n) \leq \sum_{h=0}^{\lfloor \lg n \rfloor} \left\lceil \frac{n}{2^{h+1}} \right\rceil O(h)$$

Removing the ceiling function for the asymptotic bound (since $\lceil x \rceil < x + 1$) and factoring out constants:

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

Summation Convergence

To evaluate the summation $\sum_{h=0}^{\infty} \frac{h}{2^h}$, consider the standard infinite series for $|x| < 1$:

$$\sum_{k=0}^{\infty} kx^k = \frac{x}{(1-x)^2}$$

Setting $x = 1/2$:

$$\sum_{h=0}^{\infty} \frac{h}{2^h} = \frac{1/2}{(1-1/2)^2} = \frac{1/2}{1/4} = 2$$

Final Complexity

Since the infinite sum converges to a constant (2), the finite sum is also bounded by a constant.

$$T(n) = O(n \cdot 2) = O(n)$$

Conclusion

We have proven that building a heap from an unordered array takes **linear time**, $O(n)$. This is a significant improvement over the $O(n \log n)$ approach of inserting elements one by one, making `Build-Heap` the standard method for initializing priority queues and heapsort.

2 Matrix Decompositions

Question 4. Explain the LU decomposition of a matrix using Gaussian Elimination. Clearly describe each step involved in the process.

Detailed Solution:

Introduction to LU Decomposition

LU decomposition is a factorization of a square matrix A into two triangular matrices:

$$A = L \cdot U$$

where L is a **Lower Triangular Matrix** (typically with 1s on the diagonal) and U is an **Upper Triangular Matrix**. This factorization effectively records the steps of Gaussian Elimination.

The Gaussian Elimination Process

We transform A into U via a sequence of row operations. The multipliers used to zero out elements below the diagonal are stored in L.

Initialization:

- Let $U^{(0)} = A$.
- Let L be the identity matrix I.

Step-by-Step Procedure: For each column k from 1 to $n-1$:

1. **Identify Pivot:** Let the pivot element be U_{kk}. (Assume $U_{kk} \neq 0$; otherwise, row swapping/pivoting is needed).
2. **Eliminate Entries Below Pivot:** For each row i from $k+1$ to n:
   (a) Compute the **multiplier** λ:
   $$\lambda = \frac{U_{ik}}{U_{kk}}$$
   (b) This multiplier λ represents the factor needed to cancel out the element U_{ik} using the pivot row. We store this multiplier in the lower triangular matrix:
   $$L_{ik} = \lambda$$

(c) **Update Row i:** Subtract λ times row k from row i to produce the new row i in U.

$$U_{ij} \leftarrow U_{ij} - \lambda \cdot U_{kj} \quad \text{for } j = k, \dots, n$$

Outcome

Upon completion of the loops:

- The matrix U contains the **Row Echelon Form** of A. All elements below the main diagonal are zero.
- The matrix L contains the unit diagonal (1s) and the multipliers used during elimination in the strict lower triangular part.

Example Structure

If $A = \begin{pmatrix} 2 & 1 \\ 4 & 3 \end{pmatrix}$:

1. Pivot $A_{11} = 2$. To eliminate $A_{21} = 4$, multiplier is $4/2 = 2$.
2. $L_{21} = 2$.
3. Row 2 becomes: $[4,3] - 2 \times [2,1] = [0,1]$.

Result:

$$L = \begin{pmatrix} 1 & 0 \\ 2 & 1 \end{pmatrix}, \quad U = \begin{pmatrix} 2 & 1 \\ 0 & 1 \end{pmatrix}$$

Complexity Analysis

The complexity is dominated by the nested loops in the elimination step.

- The outer loop runs n times for each pivot k.
- For each pivot, we iterate through the remaining $n-k$ rows (i loop).
- For each row, we perform subtraction across the remaining $n-k$ columns (j loop).

Total operations roughly:

$$\sum_{k=1}^{n-1} (n-k)^2 \approx \sum_{k=1}^{n} k^2 = \frac{n(n+1)(2n+1)}{6} \approx \frac{n^3}{3}$$

Thus, the time complexity of LU decomposition is $O(n^3)$.

Question 5. Solve the following recurrence relation arising from the LUP decomposition solve procedure:

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

Detailed Solution:

Context of the Recurrence

This recurrence models the computational cost of the two-step solution process for linear systems after decomposition:

1. **Forward Substitution ($Ly = Pb$):** Represented by the first summation term.
2. **Backward Substitution ($Ux = y$):** Represented by the second summation term.

Analysis of the First Term (Forward Substitution)

Let T_1 be the first summation:

$$T_1 = \sum_{i=1}^{n} \left[c_1 + \sum_{j=1}^{i-1} c_2 \right]$$

where c_1, c_2 are constants ($O(1)$). The inner sum runs $(i-1)$ times. Thus:

$$T_1 \approx \sum_{i=1}^{n} (c_1 + c_2(i-1))$$

This is an arithmetic progression involving i.

$$T_1 = c_1 n + c_2 \sum_{i=1}^{n} (i-1)$$

Using the summation formula $\sum_{k=1}^{n} k = \frac{n(n+1)}{2}$:

$$T_1 \approx c_2 \frac{n^2}{2} = O(n^2)$$

Analysis of the Second Term (Backward Substitution)

Let T_2 be the second summation:

$$T_2 = \sum_{i=1}^{n} \left[c_3 + \sum_{j=i+1}^{n} c_4 \right]$$

The inner sum runs from $j = i + 1$ to n, which is exactly $n - (i + 1) + 1 = n - i$ iterations.

$$T_2 \approx \sum_{i=1}^{n} (c_3 + c_4(n - i))$$

Let $k = n - i$. As i goes from 1 to n, k goes from $n - 1$ to 0.

$$T_2 \approx \sum_{k=0}^{n-1} (c_3 + c_4 k)$$

This is also an arithmetic progression summing to $O(n^2)$.

Total Complexity

Adding both components:

$$T(n) = T_1 + T_2 = O(n^2) + O(n^2) = O(n^2)$$

Conclusion

We have analytically derived that solving a linear system $Ax = b$ given the LUP decomposition takes $O(n^2)$ time. This highlights the efficiency of the decomposition strategy: if we need to solve for multiple vectors b, we only perform the expensive $O(n^3)$ decomposition once, and each subsequent solve is a fast $O(n^2)$ operation.

Question 6. Prove that if matrix A is non-singular, then its Schur complement is also non-singular.

Detailed Solution:

Definitions

Let A be a block partitioned matrix:

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

Assume B is non-singular (invertible). The **Schur Complement** of B in A, denoted S, is defined as:

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

LDU Decomposition of Block Matrix

We can decompose the block matrix A using block Gaussian elimination logic. Specifically, A can be factored as:

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

Let's verify the center and right matrices product:

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

Now multiply by the left matrix:

$$\begin{pmatrix} I & 0 \\ DB^{-1} & I \end{pmatrix} \begin{pmatrix} B & C \\ 0 & S \end{pmatrix} = \begin{pmatrix} B & C \\ DB^{-1}B & DB^{-1}C + S \end{pmatrix} = \begin{pmatrix} B & C \\ D & E \end{pmatrix}$$

(Since $S = E - DB^{-1}C$, so $DB^{-1}C + S = E$).

Determinant Analysis

Using the property that $\det(XY) = \det(X)\det(Y)$, we take the determinant of the decomposition:

$$\det(A) = \det(L) \cdot \det(\text{BlockDiag}) \cdot \det(U)$$

The matrices L and U are block triangular with Identity matrices on the diagonals, so their determinants are exactly 1. Thus:

$$\det(A) = 1 \cdot \det\begin{pmatrix} B & 0 \\ 0 & S \end{pmatrix} \cdot 1$$

For a block diagonal matrix, the determinant is the product of the determinants of the blocks:

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

Conclusion of Proof

We are given that A is non-singular, which means $\det(A) \neq 0$. From the equation $\det(A) = \det(B) \cdot \det(S)$, it implies that the product of the two scalars on the right side is non-zero. Therefore, it is necessary that $\det(S) \neq 0$.

Since the determinant of S is non-zero, the Schur complement S is **non-singular**.

Question 7. Prove that positive-definite matrices are suitable for LU decomposition and do not require pivoting to avoid division by zero in the recursive strategy.

Detailed Solution:

Definition of Positive-Definiteness

A symmetric real matrix A is **positive-definite (PD)** if for every non-zero vector x, the scalar $x^T Ax > 0$.

Key Properties

Two fundamental properties of positive-definite matrices are crucial for this proof:

1. **Submatrix Property:** Every leading principal submatrix of a positive-definite matrix is itself positive-definite.
2. **Determinant Property:** The determinant of any positive-definite matrix is strictly positive.

Recursive Step Analysis

LU decomposition works recursively. At the first step, we choose the pivot A_{11}. Since A_{11} is a 1×1 leading principal submatrix of A, by property (1), it must be positive-definite. For a scalar, positive-definiteness implies $A_{11} > 0$. **Result:** $A_{11} \neq 0$, so division by A_{11} is safe.

After the first step of elimination, we are left with the Schur complement S in the bottom right block ($S = E - DB^{-1}C$). A powerful theorem in linear algebra states that *the Schur complement of a positive-definite matrix is also positive-definite*.

Inductive Argument

- **Base Case:** The first pivot $A_{11} > 0$, so step 1 is safe.
- **Inductive Step:** The remaining submatrix (Schur complement) to be processed is also positive-definite. Therefore, its top-left element (the next pivot) will also be strictly positive.

Illustrative Example

Consider the symmetric positive-definite matrix $A = \begin{pmatrix} 4 & 2 \\ 2 & 5 \end{pmatrix}$.

1. **Check Pivot:** $A_{11} = 4$. Since $4 > 0$, no pivoting is needed.
2. **Elimination:** We want to eliminate $A_{21} = 2$.

$$\text{Multiplier } l_{21} = A_{21}/A_{11} = 2/4 = 0.5$$

$$\text{New } A_{22} = 5 - (0.5 \times 2) = 4$$

3. **Result:** The remaining element is 4, which is also > 0.
4. **Decomposition:**

$$L = \begin{pmatrix} 1 & 0 \\ 0.5 & 1 \end{pmatrix}, \quad U = \begin{pmatrix} 4 & 2 \\ 0 & 4 \end{pmatrix}$$

This demonstrates how the positive-definite property guarantees non-zero pivots at every step.

Conclusion

Because every pivot encountered throughout the entire recursive decomposition process is guaranteed to be strictly positive (and thus non-zero), the algorithm will never encounter a division-by-zero error. Consequently, **pivoting** (row swapping) is not required for numerical stability or solvability when A is positive-definite.

3 Graph Algorithms

Question 8. For finding an augmenting path in a graph, should Breadth First Search (BFS) or Depth First Search (DFS) be applied? Justify your answer.

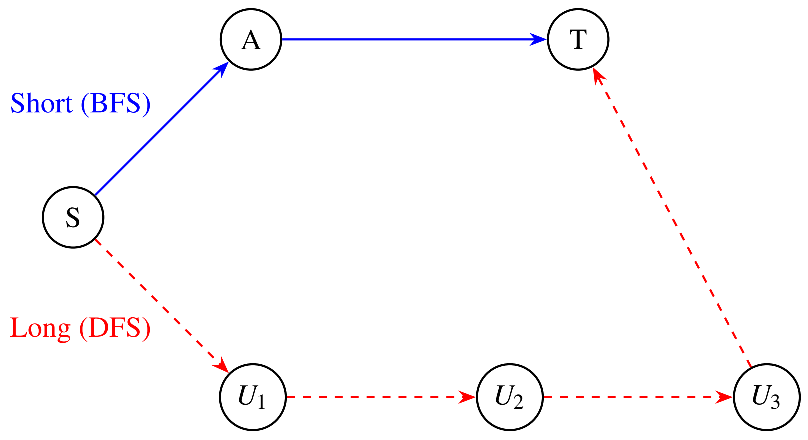
Detailed Solution:

Recommendation

Breadth First Search (BFS) should be applied. When BFS is used to find augmenting paths in the Ford-Fulkerson method, the specific implementation is known as the **Edmonds-Karp algorithm**.

Visual Comparison: BFS vs. DFS

The diagram below illustrates why BFS is superior. In the graph, DFS might explore the long path $S \to U_1 \to U_2 \to U_3 \to T$ first, whereas BFS will immediately find the direct path $S \to A \to T$.



Detailed Justification

1. **Shortest Path Guarantee:** BFS explores the graph layer by layer. Therefore, it is guaranteed to find the *shortest* augmenting path (in terms of the number of edges) from the source to the sink. DFS, on the other hand, might explore a very long, winding path before hitting the sink.

2. **Complexity Bound:** Using BFS guarantees a polynomial time complexity. It can be proven that if we always choose the shortest augmenting path, the length of the shortest path increases monotonically over time.

   - The total number of augmentations is bounded by $O(VE)$.

- Each BFS takes $O(E)$.
- Total Complexity: $O(VE^2)$.

3. **Avoiding Pathological Cases:** If DFS is used (standard Ford-Fulkerson), the algorithm depends heavily on edge capacities. In graphs with large integer capacities, DFS might choose paths that reduce flow bottlenecks by only 1 unit at a time. This can lead to pseudo-polynomial time complexity (e.g., proportional to the max capacity C), which is inefficient.

Conclusion

BFS is strictly superior to DFS for the Ford-Fulkerson method because it guarantees the shortest augmenting path property. This property transforms the algorithm from one with potentially exponential (or pseudo-polynomial) time complexity into the strongly polynomial Edmonds-Karp algorithm ($O(VE^2)$), ensuring efficient performance even for networks with large capacities.

Question 9. Explain why Dijkstra's algorithm cannot be applied to graphs with negative edge weights.

Detailed Solution:

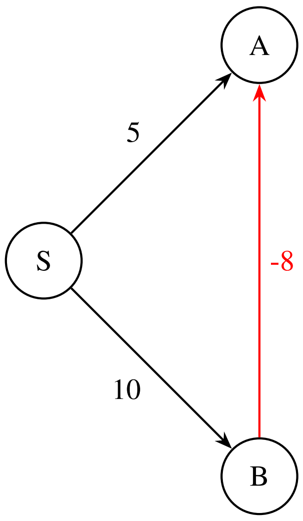
The Greedy Choice Property

Dijkstra's algorithm relies fundamentally on a greedy strategy. It maintains a set of "visited" or "finalized" nodes for which the shortest path from the source is known. The core assumption is: *"Once a node u is added to the finalized set, its shortest path distance $d[u]$ is optimal and will never change."*

This assumption holds for non-negative weights because extending a path can only increase (or keep constant) the total distance. You can never find a "shortcut" to a finalized node by going through a new, longer path.

Failure Mode with Negative Edges

When negative edges exist, this assumption collapses. Consider the following graph: $S \to A$ (cost 5), $S \to B$ (cost 10), $B \to A$ (cost -8).



Dijkstra's Execution Trace:

1. Start at S. Neighbors are $A(5)$ and $B(10)$.
2. Dijkstra picks the smallest distance: A with distance 5. It **finalizes** A.
3. It then processes B (distance 10). From B, it sees an edge to A with weight -8.
4. New path to A via B: $10+(-8)=2$.
5. This is smaller than the "finalized" distance of 5.

Conclusion

Dijkstra's algorithm finalized A prematurely. It does not go back to re-process finalized nodes. Consequently, it computes incorrect shortest path distances. For graphs with negative weights, the **Bellman-Ford algorithm** must be used, which can handle negative edges and detect negative cycles.

Question 10. Prove that every connected component of the symmetric difference of two matchings in a graph G is either a path or an even-length cycle.

Detailed Solution:

Definitions

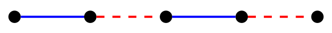
Let M_1 and M_2 be two valid matchings in a graph $G = (V,E)$. The **Symmetric Difference**, denoted $G' = M_1 \oplus M_2$, contains edges that appear in exactly one of the two matchings, but not both.

$$E(G') = (M_1 \setminus M_2) \cup (M_2 \setminus M_1)$$

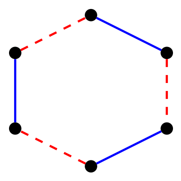
Visualization of Component Types

The diagram below shows the only two possible structures for a connected component in $M_1 \oplus M_2$: an alternating path or an alternating cycle.

Case 1: Alternating Path



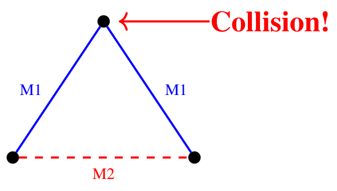
Case 2: Alternating Cycle (Even Length)



Visual Proof of Impossible Structures

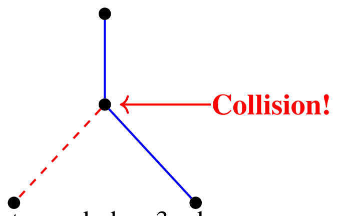
Why can't we have an odd cycle or a node with degree > 2?

Impossible Case 1: Odd Cycle



Node at top has
two M1 edges.
Violates Matching Property.

Impossible Case 2: Degree > 2



Center node has 3 edges.
Pigeonhole Principle:
At least 2 must
be M1 or M2.

Analysis of Impossible Structures

Case 1: Impossible Degree > 2

By definition, a matching is a set of edges without common vertices. For any node v:

- v is incident to at most 1 edge in M_1.
- v is incident to at most 1 edge in M_2.

The symmetric difference graph G' contains edges from both sets. Thus, the degree of v in G' is the sum of its degrees in M_1 and M_2:

$$deg_{G'}(v) \leq 1 + 1 = 2$$

Therefore, no node can have a degree of 3 or higher. This eliminates "star graphs" or any branching structures.

Case 2: Impossible Odd Cycles

If a component is a cycle, the edges must alternate between M_1 and M_2 because no two edges from the same matching can share a vertex (degree constraint). Let the edges be $e_1, e_2, \dots, e_k$.

- If $e_1 \in M_1$, then e_2 must be in M_2 (to avoid two M_1 edges at vertex v_2).
- Then e_3 must be in M_1, e_4 in M_2, and so on.

In general, edges at odd positions $(1,3,\dots)$ are in M_1 and even positions $(2,4,\dots)$ are in M_2. For a cycle to close, the last edge e_k connects back to the start of e_1. If the cycle length k were odd, then e_k would be an odd-numbered edge, meaning $e_k \in M_1$. But e_1 is also in M_1. This would mean the vertex v_1 is incident to two edges from M_1 (e_k and e_1), which violates the matching property. Thus, k must be even.

Conclusion

We have proven that every connected component in the symmetric difference must have a maximum degree of 2 and cannot form odd cycles. The only graph structures satisfying these constraints are isolated vertices, simple paths, and even-length cycles.

---

4 Complexity Classes

Question 11. Define the class Co-NP. Explain the type of problems that belong to this complexity class.

Detailed Solution:

Definition

Co-NP is a complexity class containing decision problems for which the "No" instances can be efficiently verified. Formally, a language L is in Co-NP if its complement, $\bar{L}$ (the set of strings not in L), is in the class **NP**.

In simpler terms:

- **NP Problem:** "If the answer is YES, there exists a short proof (certificate) that I can check in polynomial time."
- **Co-NP Problem:** "If the answer is NO, there exists a short proof (counter-example) that I can check in polynomial time."

Visual Relationship of Complexity Classes

The relationship between P, NP, and Co-NP is often visualized using a Venn diagram. P is contained in the intersection of NP and Co-NP.

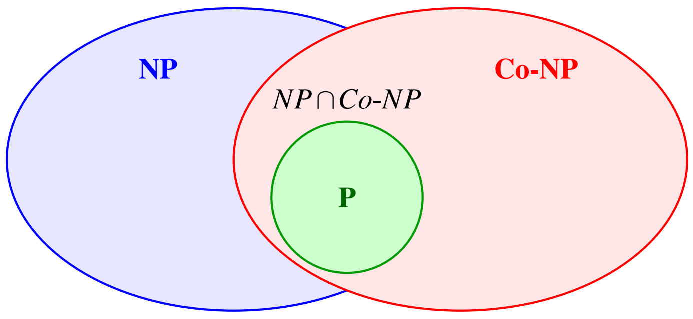


Diagram assumes $P \neq NP$ and $NP \neq Co\text{-}NP$

Type of Problems

Co-NP problems typically involve properties that must hold for **all** possible structures or assignments (Universality).

1. **TAUTOLOGY (Logic):**
   - *Problem:* Given a boolean formula ϕ, is it true for **every** possible assignment of truth values to variables?
   - *Why Co-NP:* If the answer is NO, there exists at least one assignment where ϕ is False. This single assignment serves as a succinct "counter-example" (certificate) that can be verified easily.
2. **UNSAT (Logic):**
   - *Problem:* Given a boolean formula ϕ, is it impossible to satisfy? (i.e., Is it false for all assignments?)
   - *Why Co-NP:* This is the direct complement of SAT. If the answer is NO (meaning it IS satisfiable), we can present a satisfying assignment as proof.
3. **GRAPH NON-ISOMORPHISM:**
   - *Problem:* Given two graphs G_1 and G_2, are they **not** isomorphic?
   - *Why Co-NP:* The complement problem is Graph Isomorphism (in NP). If the answer to Non-Isomorphism is NO (meaning they ARE isomorphic), the certificate is simply the mapping (bijection) between the vertices.
4. **NON-HAMILTONIAN GRAPH:**
   - *Problem:* Given a graph G, is it true that there is **no** Hamiltonian cycle?
   - *Why Co-NP:* If the answer is NO (meaning there IS a cycle), the certificate is simply the sequence of vertices forming the cycle.

Conclusion

The class Co-NP complements NP. While NP characterizes problems where "Yes" answers are easy to verify (existence proofs), Co-NP characterizes problems where "No" answers are easy to verify (universal truths). Most computer scientists believe $NP \neq Co\text{-}NP$, meaning that just because we can verify a solution quickly doesn't mean we can verify the *absence* of a solution quickly.

Question 12. Given a Boolean circuit instance whose output evaluates to true, explain how the correctness of the result can be verified in polynomial time using Depth First Search (DFS).

Detailed Solution:

Problem Context

The **Circuit Value Problem** asks given a boolean circuit (a Directed Acyclic Graph of gates) and inputs, what is the output? We are discussing the *verification* aspect. However, since the problem is in **P**, verification is essentially just re-running the computation.

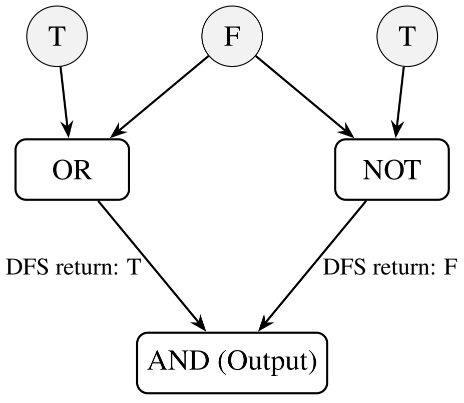
Verification Algorithm using DFS

We can verify the circuit's output by re-evaluating the circuit. Since a circuit is a DAG:

1. **Graph Representation:** Treat gates as nodes and wires as directed edges.
2. **Recursive DFS:** We define a function `Evaluate(gate)`:
   - If `gate` is an Input, return its assigned value.
   - If `gate` has been visited/memoized, return the stored value.
   - Recursively call `Evaluate` on all inputs to this gate.
   - Apply the logic operation (AND, OR, NOT) to the results.
   - Store and return the result.

Visualizing Circuit Evaluation via DFS

The diagram below shows a simple boolean circuit. A DFS traversal would start at the output node and recursively request values from inputs (leaf nodes), computing the logic at each step.



Final Result: False

Complexity Analysis

- **Vertices (V):** Each gate is a vertex.
- **Edges (E):** Each wire is an edge.

Using DFS with memoization (to handle fan-out, where one gate's output feeds multiple inputs), we visit each node and traverse each edge exactly once. The logic computation at each node takes $O(1)$ time. Total Time Complexity: $O(V+E)$.

Conclusion

Since the size of the circuit description is proportional to $V+E$, the verification runs in linear time relative to the input size. This is strictly polynomial time.

Question 13. Is the 3-SAT (3-CNF-SAT) problem NP-Hard? Justify your answer.

Detailed Solution:

Answer

Yes, the 3-SAT problem is **NP-Hard**.

Detailed Explanation

1. Introduction

The Boolean Satisfiability Problem (SAT) was the first problem proven to be **NP-Complete** (Cook-Levin Theorem). The 3-SAT problem is a specific variant of SAT where the structure of the boolean formula is restricted.

2. Definition of 3-SAT

For a problem to be classified as 3-SAT, the boolean formula must satisfy two conditions:

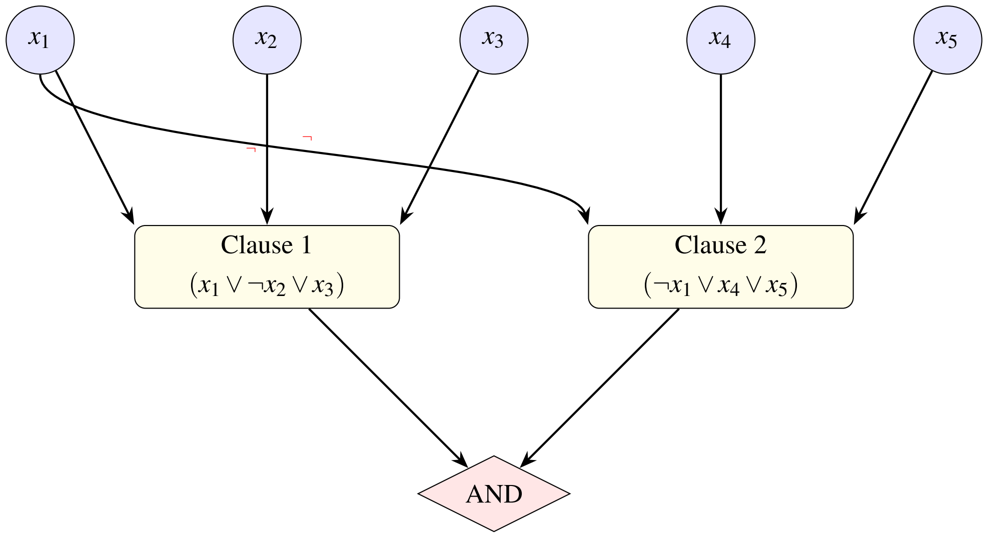
1. It must be in **Conjunctive Normal Form (CNF)**, meaning it is an AND of multiple clauses.
2. Each clause must contain **exactly three literals** (a literal is a variable or its negation).

Example Instance: Consider the following formula with 5 variables and 2 clauses:

$$\phi = (x_1 \vee \neg x_2 \vee x_3) \wedge (\neg x_1 \vee x_4 \vee x_5)$$

3. Visual Structure

The diagram below illustrates the logic flow of the specific example above. Inputs are variables, which form clauses (OR logic), and the system is satisfied only if the global AND is true.


x_1
x_2
x_3
x_4
x_5
$\neg$
$\neg$
Clause 1
$(x_1 \vee \neg x_2 \vee x_3)$
Clause 2
$(\neg x_1 \vee x_4 \vee x_5)$
AND

Result: True iff Clause 1 = True AND Clause 2 = True

4. Justification of NP-Hardness

To classify 3-SAT as NP-Hard, we use the concept of **Polynomial-Time Reduction** ($\leq_p$).

- **Premise:** We know that the general SAT problem is **NP-Complete**. This means every problem in NP can be reduced to SAT.
- **Reduction:** It has been proven that any general SAT instance (where clauses can have arbitrary length) can be transformed into an equivalent 3-SAT instance in polynomial time. This is done by splitting long clauses into chains of size-3 clauses using auxiliary variables.
- **Logic:** Since $SAT \in$ NP-Hard and $SAT \leq_p 3\text{-}SAT$, transitivity implies that 3-SAT must be at least as hard as SAT.

Conclusion

Therefore, **3-SAT is NP-Hard**. Furthermore, since a given assignment of truth values can be verified against a 3-CNF formula in polynomial time, 3-SAT is also in NP. Being both in NP and NP-Hard makes it **NP-Complete**.

Question 14. Is the 2-SAT problem NP-Hard? Can it be solved in polynomial time? Explain your reasoning.

Detailed Solution:

Answer

No, 2-SAT is **not NP-Hard** (assuming $P \neq NP$). It is in the complexity class **P** and can be solved in linear time.

Polynomial Time Solution Strategy

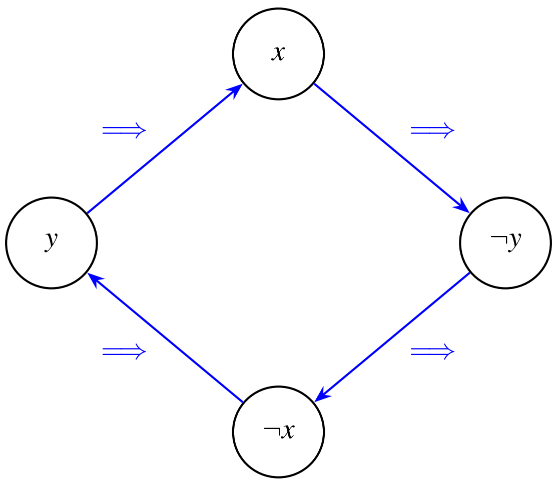
The 2-SAT problem (where every clause has exactly 2 literals) can be modeled using a directed graph called the **Implication Graph**.

Construction: For every clause $(A \vee B)$, we add two directed edges representing the logical implications:

1. If A is False, B must be True: $\neg A \implies B$.
2. If B is False, A must be True: $\neg B \implies A$.

Visualizing the Implication Graph

The graph below illustrates a scenario where a variable implies its own negation and vice versa via a cycle (Strongly Connected Component). This condition proves unsatisfiability.



Unsatisfiable Condition:
Path exists $x \rightsquigarrow \neg x$ AND $\neg x \rightsquigarrow x$.
They are in the same SCC.

Satisfiability Condition: The formula is unsatisfiable if and only if there exists a variable x such that there is a path from x to $\neg x$ **AND** a path from $\neg x$ to x. In graph theory terms, this means x and $\neg x$ belong to the same **Strongly Connected Component (SCC)**.

Algorithm

1. Build the implication graph. Nodes are literals $(x_i, \neg x_i)$.
2. Run an SCC finding algorithm (like **Kosaraju's** or **Tarjan's Algorithm**).
3. Check if any variable x_i and its negation $\neg x_i$ are in the same SCC.
4. If yes, return "Unsatisfiable". If no, return "Satisfiable".

Complexity

Building the graph takes linear time in the number of clauses. Finding SCCs takes $O(V+E)$ time. Thus, 2-SAT is solvable in $O(N+M)$ time, which is polynomial. This sharp contrast with 3-SAT (which is NP-Hard) is a famous example of the "boundary" of tractability.

Conclusion

The ability to map 2-SAT to a graph reachability problem allows us to bypass exponential search. Since graph connectivity (specifically SCCs) can be solved in linear time, 2-SAT resides firmly in P, unlike its general counterpart SAT or 3-SAT.