ADSA Assignment

MTech CSE 1st Semester

IIIT Bhubaneswar

Name: Bhuvan Kumar
Student ID: A125003

Question 1

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

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

Solution

We are given the recurrence relation for the Heapify operation:

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

Let us use the substitution method to solve this recurrence. Assume the constant work at each level is c. Then:

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

The base case is reached when $\left(\frac{2}{3}\right)^k n = 1$, which gives:

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

Taking logarithm on both sides:

$$k = \log_{3/2} n = \frac{\log n}{\log(3/2)} = O(\log n)$$

Substituting back:

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

Therefore, the time complexity of the recursive Heapify operation is $O(\log n)$.

Question 2

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

Solution

Consider a binary heap represented as an array with 1-based indexing. For any node at index i:

- Left child is at index $2i$
- Right child is at index $2i + 1$

A node is classified as a leaf node if it has no children. This occurs when both $2i > n$ and $2i + 1 > n$, which simplifies to $2i > n$.

From the inequality $2i > n$, we obtain:

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

Since indices must be integers, the smallest index satisfying this condition is:

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

Conversely, for any node at index $i \leq \lfloor n/2 \rfloor$, we have $2i \leq n$, meaning the node has at least a left child and is therefore an internal node.

Hence, all leaf nodes are located at indices from $\lfloor n/2 \rfloor + 1$ to n.

Question 3

Part (a)

Show that in any heap containing n elements, the number of nodes at height h is at most: $\lceil n/2^{h+1} \rceil$

Solution (a)

In a binary heap, the height of a node is defined as the length of the longest path from that node down to a leaf.

Consider nodes at height h. Each such node must have a subtree rooted at it containing at least 2^h nodes (since a complete tree of height h has 2^h leaves).

Since the total number of nodes in the heap is n, and nodes at height h have non-overlapping subtrees each containing at least 2^h nodes, we can have at most:

$$\frac{n}{2^h} \text{ such nodes}$$

However, to account for the fact that these nodes are internal nodes at height h (not the leaves themselves), we consider that each node at height h contributes to the structure above the 2^h descendants. The tighter bound considering the complete binary tree structure gives:

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

This accounts for the geometric distribution of nodes across different heights in a heap.

Part (b)

Using the above result, prove that the time complexity of the Build-Heap algorithm is $O(n)$.

Solution (b)

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

From part (a), we know that the number of nodes at height h is at most $\lceil n/2^{h+1} \rceil$.

Since Heapify on a node at height h takes $O(h)$ time, the total time complexity is:

$$\begin{aligned} T(n) &= \sum_{h=0}^{\lfloor \log n \rfloor} \left\lceil \frac{n}{2^{h+1}} \right\rceil \cdot O(h) \\ &= O\left(n \sum_{h=0}^{\lfloor \log n \rfloor} \frac{h}{2^{h+1}} \right) \\ &= O\left(n \sum_{h=0}^{\infty} \frac{h}{2^h} \right) \end{aligned}$$

The infinite series $\sum_{h=0}^{\infty} \frac{h}{2^h}$ is a well-known convergent series that equals 2.

Therefore:

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

Thus, the Build-Heap algorithm has linear time complexity.

Question 4

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

Solution

LU decomposition factors a square matrix A into a product of two matrices:

$$A = LU$$

where:

- L is a lower triangular matrix with ones on the diagonal
- U is an upper triangular matrix

Process using Gaussian Elimination:
Step 1: Start with matrix A and initialize L as an identity matrix and U as A.
Step 2: For each column k from 1 to $n-1$:

- Identify the pivot element U_{kk}
- For each row i below row k (i.e., $i > k$):
  - Calculate the multiplier: $\ell_{ik} = U_{ik}/U_{kk}$
  - Store this multiplier in L_{ik}
  - Update row i of U by subtracting ℓ_{ik} times row k

Step 3: The resulting matrix U is upper triangular, and L contains all the multipliers used during elimination.

Algorithm 1 LU Decomposition via Gaussian Elimination

Require: Square matrix $A \in \mathbb{R}^{n \times n}$
Ensure: Lower triangular L and upper triangular U such that $A = LU$

```
Initialize L ← I_n, U ← A
for k = 1 to n − 1 do
    for i = k + 1 to n do
        L_ik ← U_ik / U_kk
        for j = k to n do
            U_ij ← U_ij − L_ik · U_kj
        end for
    end for
end for
return L, U
```

The key insight is that each elimination step in Gaussian Elimination corresponds to multiplication by a unit lower triangular matrix, and the product of these matrices forms L.

Question 5

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

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

Solution

Let's analyze each summation separately.

First summation:

$$\begin{aligned}
\sum_{i=1}^{n} \left[O(1) + \sum_{j=1}^{i-1} O(1) \right] &= \sum_{i=1}^{n} [c + (i-1)c] \\
&= \sum_{i=1}^{n} c \cdot i \\
&= c \sum_{i=1}^{n} i \\
&= c \cdot \frac{n(n+1)}{2} \\
&= O(n^2)
\end{aligned}$$

Second summation:

$$\begin{aligned}
\sum_{i=1}^{n} \left[O(1) + \sum_{j=i+1}^{n} O(1) \right] &= \sum_{i=1}^{n} [c + (n-i)c] \\
&= \sum_{i=1}^{n} c(n-i+1) \\
&= c \sum_{i=1}^{n} (n-i+1) \\
&= c \sum_{k=1}^{n} k \quad \text{(substituting } k = n-i+1\text{)} \\
&= c \cdot \frac{n(n+1)}{2} \\
&= O(n^2)
\end{aligned}$$

Total complexity:

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

Therefore, the time complexity of the LUP decomposition solve procedure is $O(n^2)$.

Question 6

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

Solution

Consider a non-singular matrix A partitioned as:

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

where B is a non-singular square submatrix.

The Schur complement of B in A is defined as:

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

Proof by contradiction:

Assume that the Schur complement S is singular. Then there exists a non-zero vector $\mathbf{y}$ such that:

$$S\mathbf{y} = \mathbf{0}$$

$$(E - DB^{-1}C)\mathbf{y} = \mathbf{0}$$

Now construct a vector $\mathbf{x} = \begin{pmatrix} \mathbf{x}_1 \\ \mathbf{x}_2 \end{pmatrix}$ where:

$$\mathbf{x}_1 = -B^{-1}C\mathbf{y}, \quad \mathbf{x}_2 = \mathbf{y}$$

Since $\mathbf{y} \neq \mathbf{0}$, we have $\mathbf{x} \neq \mathbf{0}$.

Computing $A\mathbf{x}$:

$$\begin{aligned} A\mathbf{x} &= \begin{pmatrix} B & C \\ D & E \end{pmatrix} \begin{pmatrix} -B^{-1}C\mathbf{y} \\ \mathbf{y} \end{pmatrix} \\ &= \begin{pmatrix} -BB^{-1}C\mathbf{y} + C\mathbf{y} \\ -DB^{-1}C\mathbf{y} + E\mathbf{y} \end{pmatrix} \\ &= \begin{pmatrix} -C\mathbf{y} + C\mathbf{y} \\ (E - DB^{-1}C)\mathbf{y} \end{pmatrix} \\ &= \begin{pmatrix} \mathbf{0} \\ S\mathbf{y} \end{pmatrix} \\ &= \begin{pmatrix} \mathbf{0} \\ \mathbf{0} \end{pmatrix} \end{aligned}$$

This shows that $A\mathbf{x} = \mathbf{0}$ for some non-zero vector $\mathbf{x}$, which contradicts the assumption that A is non-singular.

Therefore, the Schur complement S must be 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.

Solution

Let A be a symmetric positive-definite matrix. By definition, for any non-zero vector $\mathbf{x}$:

$$\mathbf{x}^T A\mathbf{x} > 0$$

Key Properties:

1. All leading principal minors of a positive-definite matrix are positive.
2. All eigenvalues of a positive-definite matrix are positive.
3. All diagonal elements of a positive-definite matrix are positive.

In the recursive LU decomposition, at each step we partition the matrix as:

$$A = \begin{pmatrix} \alpha & \mathbf{v}^T \\ \mathbf{v} & B \end{pmatrix}$$

where α is the pivot element (top-left entry).

Since A is positive-definite, the 1×1 leading principal minor, which is simply α, must be positive:

$$\alpha > 0$$

This ensures that division by α in the elimination step is always valid (no division by zero).

After eliminating the first row and column, the remaining submatrix (Schur complement) is:

$$S = B - \frac{1}{\alpha}\mathbf{v}\mathbf{v}^T$$

Claim: S is also positive-definite.

Proof: For any non-zero vector $\mathbf{y}$, define $\mathbf{z} = \begin{pmatrix} -\mathbf{v}^T\mathbf{y}/\alpha \\ \mathbf{y} \end{pmatrix}$. Then:

$$\mathbf{z}^T A\mathbf{z} = \mathbf{y}^T S\mathbf{y} > 0$$

Since this recursive property holds at every step, each pivot element is guaranteed to be positive and non-zero. Therefore:

- No pivoting is required
- LU decomposition can proceed without encountering division by zero
- The decomposition is numerically stable

Thus, positive-definite matrices are ideal candidates for LU decomposition without pivoting.

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.

Solution

Answer: Breadth First Search (BFS) should be used for finding augmenting paths.
Justification:

1. **Shortest Path Guarantee:** BFS finds the shortest augmenting path in terms of the number of edges. This is crucial because using shortest paths leads to better algorithmic performance.

2. **Monotonic Progress:** When we repeatedly find and augment along shortest augmenting paths, the length of the shortest augmenting path increases monotonically. This property is exploited by efficient matching algorithms.

3. **Polynomial Time Complexity:** The Hopcroft-Karp algorithm for maximum bipartite matching uses BFS to find multiple vertex-disjoint augmenting paths of the same length simultaneously. This approach achieves $O(\sqrt{V} \cdot E)$ time complexity.

4. **Avoiding Inefficiency:** DFS may find arbitrarily long augmenting paths, which can lead to:

   - More iterations needed to reach maximum matching
   - Potential for exponential behavior in worst cases
   - Less predictable algorithm performance

5. **Level-Based Processing:** BFS naturally organizes the graph into levels, which allows for phase-based augmentation strategies that are more efficient than arbitrary path selection.

Example: In the Ford-Fulkerson method for maximum flow, using BFS (resulting in the Edmonds-Karp algorithm) guarantees $O(VE^2)$ complexity, while DFS can potentially run in exponential time.

Therefore, BFS is the preferred search strategy for finding augmenting paths in matching and flow algorithms.

Question 9

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

Solution

Dijkstra's algorithm is a greedy algorithm that makes irrevocable decisions based on local information. The fundamental assumption is:

Greedy Choice Property: Once a vertex is removed from the priority queue (with minimum tentative distance), its shortest path distance from the source is finalized and will not change.

This property holds **only when all edge weights are non-negative**.

Why Negative Weights Break Dijkstra's Algorithm:

1. **Premature Finalization:** When Dijkstra extracts a vertex v with distance $d[v]$, it assumes this is the shortest possible distance to v. However, if negative edges exist, a path discovered later might have a shorter total distance.

2. **Violated Monotonicity:** With non-negative edges, any extension of a path can only increase (or maintain) its length. Negative edges violate this monotonicity, allowing later paths to be shorter.

3. **Counter-Example:** Consider this graph:

   - Vertices: S, A, B
   - Edges: $S \to A$ (weight 5), $S \to B$ (weight 2), $B \to A$ (weight -4)
   - Shortest path from S to A: $S \to B \to A$ with distance $2 + (-4) = -2$

   Dijkstra would extract B first (distance 2), then A (distance 5), missing the shorter path $S \to B \to A$ with distance -2 because A was already finalized.

4. **Incorrect Results:** The algorithm produces wrong shortest path distances and cannot detect negative-weight cycles.

Alternative Algorithms:

- **Bellman-Ford Algorithm:** Handles negative edge weights correctly and can detect negative cycles. Time complexity: $O(VE)$.
- **SPFA (Shortest Path Faster Algorithm):** An optimization of Bellman-Ford with better average-case performance.

Conclusion: Dijkstra's greedy strategy inherently requires non-negative edge weights to guarantee correctness. With negative edges, the algorithm's fundamental assumption is violated, leading to incorrect results.

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.

Solution

Let M_1 and M_2 be two matchings in graph G. The symmetric difference is defined as:

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

This consists of all edges that belong to exactly one of the two matchings.
Degree Analysis:
Consider the subgraph $H = (V, M_1 \oplus M_2)$ induced by the symmetric difference.
For any vertex $v \in V$:

- v is incident to at most one edge from M_1 (since M_1 is a matching)
- v is incident to at most one edge from M_2 (since M_2 is a matching)
- Therefore, $\deg_H(v) \leq 2$

Connected Component Structure:
Since every vertex in H has degree at most 2, each connected component must be one of:

1. An isolated vertex (degree 0)
2. A simple path (with two endpoints of degree 1 and internal vertices of degree 2)
3. A simple cycle (all vertices of degree 2)

Alternating Property:
Within each connected component, edges must alternate between M_1 and M_2 because:

- No two edges from the same matching can be adjacent (matching property)
- All edges in the component come from $M_1 \oplus M_2$

Cycle Length:
If a connected component is a cycle:

- Edges alternate between M_1 and M_2
- Number of edges from M_1 equals number of edges from M_2 (to complete the cycle)
- Total number of edges $= 2k$ for some integer k
- Therefore, the cycle has even length

Path Properties:
If a connected component is a path:

- Edges alternate between M_1 and M_2

- The path may have odd or even length
- Both endpoints have degree 1

Conclusion:
Every connected component of $M_1 \oplus M_2$ has maximum degree 2 and exhibits an alternating structure. Therefore, each component is either:

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

This property is fundamental in matching theory and is used extensively in algorithms for finding maximum matchings.

Question 11

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

Solution

Definition of Co-NP:

The complexity class Co-NP (Complement of NP) consists of all decision problems whose complements are in NP.

Formally: A language L belongs to Co-NP if and only if $\overline{L} \in$ NP, where $\overline{L}$ is the complement of L.

Equivalent Characterization:

A problem Π is in Co-NP if for every NO-instance of Π, there exists a certificate that can be verified in polynomial time to prove that the answer is indeed NO.

Relationship with NP:

- **NP:** Problems where YES-instances have efficiently verifiable certificates
- **Co-NP:** Problems where NO-instances have efficiently verifiable certificates
- **P $\subseteq$ NP $\cap$ Co-NP:** Every problem in P is in both NP and Co-NP
- **Open Question:** Whether NP = Co-NP is unknown (most believe they are different)

Types of Problems in Co-NP:

1. **Tautology (TAUT):** Given a Boolean formula ϕ, is ϕ true for all possible truth assignments?
   - To show ϕ is not a tautology (NO-instance), provide an assignment that makes ϕ false
   - This assignment can be verified in polynomial time
2. **Prime Numbers:** Is a given number n composite?
   - To prove n is composite (NO-instance for primality), provide factors
   - Verification takes polynomial time
3. **Unsatisfiability:** Given a Boolean formula ϕ, is ϕ unsatisfiable?
   - This is the complement of SAT
   - Proving unsatisfiability generally requires showing all assignments fail
4. **Non-Hamiltonian Graphs:** Does a graph have no Hamiltonian cycle?
   - Complement of the Hamiltonian Cycle problem
   - Proving absence is generally harder than proving existence

Characteristics of Co-NP Problems:

- Focus on proving negative results (showing something doesn't exist)
- Often involve universal quantification ("for all")
- Tend to be at least as hard as their NP counterparts
- Many Co-NP-complete problems are complements of NP-complete problems

Example - TAUT in Detail:
For the tautology problem:

- **YES-instance:** $\phi \equiv (x \vee \neg x)$ is a tautology
- **NO-instance:** $\phi \equiv (x \wedge y)$ is not a tautology
- **Certificate for NO:** Assignment $x = 0, y = 0$ makes ϕ false
- **Verification:** Check that this assignment evaluates ϕ to false (polynomial time)

Therefore, Co-NP captures the complexity of problems where disproving a statement can be done efficiently with an appropriate certificate.

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).

Solution

Boolean Circuit Representation:

A Boolean circuit can be modeled as a directed acyclic graph (DAG) where:

- Each node represents a logic gate (AND, OR, NOT, etc.) or an input variable
- Edges represent the flow of Boolean values from inputs to outputs
- There is a single designated output gate

Verification Problem:

Given:

- A Boolean circuit C
- An assignment to input variables (certificate)
- Claim that the circuit evaluates to true

Task: Verify in polynomial time that the circuit indeed outputs true with the given input assignment.

DFS-Based Verification Algorithm:

The verification process uses DFS to evaluate the circuit bottom-up:

Algorithm 2 Circuit Verification using DFS

Require: Boolean circuit C, input assignment σ
Ensure: Returns true if circuit evaluates to true, false otherwise

```
Initialize all gate values as uncomputed
Start DFS from the output gate
function EvaluateGate(gate g)
    if g is an input gate then
        return value from assignment σ
    end if
    if g has been evaluated then
        return cached value of g
    end if
    Let in_1, in_2, ..., in_k be the input gates to g
    for each input gate in_i do
        val_i ← EvaluateGate(in_i)
    end for
    result ← apply gate operation of g to (val_1, ..., val_k)
    Cache result for gate g
    return result
end function
return EvaluateGate(output gate)
```

Correctness:

1. **Complete Evaluation:** DFS ensures every gate needed to compute the output is evaluated exactly once.
2. **Proper Dependencies:** A gate is evaluated only after all its input gates have been evaluated (post-order traversal).
3. **Acyclic Property:** Since the circuit is a DAG, there are no circular dependencies, and DFS terminates.
4. **Memoization:** Caching gate values prevents redundant computation when gates have multiple outgoing edges.

Time Complexity Analysis:
Let n be the number of gates in the circuit and m be the number of edges.

- Each gate is visited at most once: $O(n)$
- Each edge is traversed at most once: $O(m)$
- Evaluating each gate takes constant time (for standard Boolean operations)
- Total time: $O(n + m)$

Since the circuit has polynomial size in the input (number of input variables), the verification can be completed in time polynomial in the size of the problem instance.

Why This Makes the Problem in NP:
The Circuit-SAT problem (determining if there exists an input assignment that makes the circuit output true) is in NP because:

- **Certificate:** An assignment to all input variables
- **Verification:** Use the DFS algorithm above to evaluate the circuit
- **Polynomial Time:** Verification takes $O(n + m)$ time

Therefore, given a Boolean circuit and an input assignment, DFS provides an efficient polynomial-time method to verify that the circuit evaluates to true.

Question 13

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

Solution

Answer: Yes, the 3-SAT problem is NP-Hard (and in fact, NP-Complete).

Problem Definition:

3-SAT is the problem of determining whether a Boolean formula in Conjunctive Normal Form (CNF) with exactly 3 literals per clause is satisfiable.

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

Proof that 3-SAT is NP-Hard:

Step 1: Show 3-SAT $\in$ NP

3-SAT is in NP because:

- **Certificate:** A truth assignment to all variables
- **Verification:** Evaluate each clause with the assignment
- **Time:** If there are m clauses, verification takes $O(m)$ time

Step 2: Prove NP-Hardness via Reduction

We show that SAT (general CNF satisfiability) reduces to 3-SAT in polynomial time: SAT $\leq_p$ 3-SAT.

Since SAT is NP-Complete (Cook-Levin theorem), this proves 3-SAT is NP-Hard.

Reduction Construction:

For each clause C in the SAT instance, we convert it to an equivalent set of 3-literal clauses:

1. **If $|C| = 1$:** Clause is (l_1)
   - Create: $(l_1 \vee y \vee z) \wedge (l_1 \vee y \vee \neg z) \wedge (l_1 \vee \neg y \vee z) \wedge (l_1 \vee \neg y \vee \neg z)$
   - New variables y, z force l_1 to be true
2. **If $|C| = 2$:** Clause is $(l_1 \vee l_2)$
   - Create: $(l_1 \vee l_2 \vee y) \wedge (l_1 \vee l_2 \vee \neg y)$
   - New variable y ensures equivalence
3. **If $|C| = 3$:** Clause is $(l_1 \vee l_2 \vee l_3)$
   - Keep as is: $(l_1 \vee l_2 \vee l_3)$
4. **If $|C| > 3$:** Clause is $(l_1 \vee l_2 \vee \cdots \vee l_k)$ where $k > 3$
   - Introduce new variables $y_1, y_2, \ldots, y_{k-3}$

- Create clauses:

$$
\begin{aligned}
&(l_1 \vee l_2 \vee y_1) \\
&(\neg y_1 \vee l_3 \vee y_2) \\
&(\neg y_2 \vee l_4 \vee y_3) \\
&\vdots \\
&(\neg y_{k-4} \vee l_{k-2} \vee y_{k-3}) \\
&(\neg y_{k-3} \vee l_{k-1} \vee l_k)
\end{aligned}
$$

- This chain ensures the original clause is satisfied iff at least one of the new clauses is satisfied

Correctness of Reduction:

- The transformation preserves satisfiability
- Each clause is converted in polynomial time
- Number of new clauses is linear in the size of the original clause
- Total transformation time is polynomial

Conclusion:
Since:

1. 3-SAT $\in$ NP (can verify solutions in polynomial time)
2. SAT $\leq_p$ 3-SAT (polynomial-time reduction from NP-Complete problem)
3. Therefore, 3-SAT is NP-Complete

Being NP-Complete means 3-SAT is both NP-Hard and in NP. Thus, 3-SAT is NP-Hard.

Question 14

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

Solution

Answer: No, 2-SAT is not NP-Hard. It can be solved in polynomial time, specifically in $O(n+m)$ time where n is the number of variables and m is the number of clauses.

Problem Definition:

2-SAT is the problem of determining whether a Boolean formula in CNF with at most 2 literals per clause is satisfiable.

Example: $\phi = (x_1 \vee \neg x_2) \wedge (\neg x_1 \vee x_3) \wedge (x_2 \vee \neg x_3)$

Why 2-SAT is in P:

Key Insight: Every 2-SAT clause $(a \vee b)$ can be rewritten as two implications:

- $(a \vee b) \equiv (\neg a \Rightarrow b) \wedge (\neg b \Rightarrow a)$

This allows us to model the 2-SAT instance as an implication graph.

Implication Graph Construction:

For a 2-SAT formula with variables $x_1, x_2, \ldots, x_n$:

1. **Vertices:** Create a vertex for each literal: $x_1, \neg x_1, x_2, \neg x_2, \ldots, x_n, \neg x_n$
2. **Edges:** For each clause $(l_i \vee l_j)$, add two directed edges:
   - $\neg l_i \to l_j$
   - $\neg l_j \to l_i$

Satisfiability Criterion:

The 2-SAT formula is satisfiable if and only if for every variable x_i:

x_i and $\neg x_i$ do not belong to the same strongly connected component (SCC)

Intuition: If x_i and $\neg x_i$ are in the same SCC, then $x_i \Rightarrow \neg x_i$ and $\neg x_i \Rightarrow x_i$, which creates a contradiction.

Algorithm:

Algorithm 3 2-SAT Solver

Require: 2-SAT formula ϕ with variables $x_1, \ldots, x_n$
Ensure: Returns SAT if satisfiable, UNSAT otherwise

```
Construct implication graph G from φ
Find all strongly connected components of G using Kosaraju's or Tarjan's algorithm
for each variable x_i do
    if x_i and ¬x_i are in the same SCC then
        return UNSAT
    end if
end for
return SAT
```

Finding a Satisfying Assignment:

If the formula is satisfiable, we can construct a satisfying assignment:

1. Compute SCCs and perform topological sort on the condensation graph
2. Process SCCs in reverse topological order
3. For each unassigned variable x_i:
    - If $x_i s SCC comes after \neg x_i s SCC in topological order, set \mathrm{x}_i$ = true
    - Otherwise, set x_i = false

Time Complexity Analysis:

- **Graph Construction:** $O(m)$ time for m clauses
- **SCC Computation:** $O(V+E) = O(n+m)$ using Tarjan's or Kosaraju's algorithm
    - Vertices: $O(n)$ (two literals per variable)
    - Edges: $O(m)$ (two edges per clause)
- **Checking Consistency:** $O(n)$ time
- **Total:** $O(n + m)$

Example:
Consider $\phi = (x_1 \vee x_2) \wedge (\neg x_1 \vee x_3) \wedge (\neg x_2 \vee \neg x_3) \wedge (\neg x_3 \vee x_1)$
Implication edges:

- $\neg x_1 \to x_2$, $\neg x_2 \to x_1$
- $x_1 \to x_3$, $\neg x_3 \to \neg x_1$
- $x_2 \to \neg x_3$, $x_3 \to \neg x_2$
- $x_3 \to x_1$, $\neg x_1 \to \neg x_3$

By computing SCCs, we can determine if there's a variable whose positive and negative literals are in the same SCC. If not, the formula is satisfiable.

Conclusion:

Since 2-SAT can be solved in polynomial time $O(n+m)$, it belongs to the complexity class P. Therefore:

- 2-SAT is not NP-Hard (unless P = NP)
- 2-SAT is efficiently solvable
- This contrasts sharply with 3-SAT, which is NP-Complete

The boundary between tractable and intractable SAT problems lies between 2-SAT and 3-SAT.