Homework Assignment

Advanced Data Structures and Algorithms

Name: Biprajit Roy
Roll No: A125004
Department of Computer Science

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

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

Let the constant amount of work performed at each recursive call be $c > 0$. Then the recurrence can be rewritten as

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

Step 1: Repeated Expansion

Expanding the recurrence repeatedly, we obtain

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

Step 2: Depth of Recursion

The recursion terminates when the problem size becomes constant, say 1. Thus,

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

Taking logarithms on both sides,

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

which gives

$$k \log\left(\frac{2}{3}\right) + \log n = 0.$$

Solving for k,

$$k = \frac{-\log n}{\log(2/3)} = \frac{\log n}{\log(3/2)}.$$

Since $\log(3/2)$ is a constant,

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

Step 3: Total Cost

Substituting the value of k into the expanded recurrence,

$$T(n) = T(1) + kc.$$

Since $T(1)$ and c are constants, we obtain

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

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

Conclusion

The recursive Heapify operation reduces the input size by a constant factor at each step and performs constant work per level. Therefore, the overall time complexity of Heapify is logarithmic in the input size.

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

A binary heap is a complete binary tree stored in an array. For an element stored at index i:

- The left child is at index $2i$
- The right child is at index $2i + 1$

Step 1: Condition for a node to have children

A node at index i has at least one child if

$$2i \leq n.$$

If

$$2i > n,$$

then the node has no children and is therefore a leaf node.

Step 2: Determine the indices of leaf nodes

From the inequality

$$2i > n,$$

we obtain

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

Since array indices are integers, this means

$$i \geq \lfloor n/2 \rfloor + 1.$$

Step 3: Range of leaf nodes

The largest possible index in the heap array is n. Hence, all indices satisfying

$$\lfloor n/2 \rfloor + 1 \leq i \leq n$$

correspond to nodes with no children.

Conclusion

All leaf nodes in a binary heap stored as an array of size n are located at indices from

$$\boxed{\lfloor n/2 \rfloor + 1 \text{ to } n}.$$

Question 3

(a) Show that in any heap containing n elements, the number of nodes at height h is at most

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

Solution

A binary heap is a complete binary tree. In such a tree, the number of nodes decreases as the height from the leaves increases.

Step 1: Relationship between height and number of nodes

In a binary tree, the maximum number of nodes at height h is bounded by the total number of nodes divided by the number of nodes in a complete subtree of height h.

Each level higher in the tree has at most half the number of nodes of the level below it.

Step 2: Upper bound on nodes at height h

At height h, the number of nodes is at most

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

Conclusion

Hence, in a heap containing n elements, the number of nodes at height h is at most

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

(b) Using the above result, prove that the time complexity of the `BUILD-HEAP` algorithm is $O(n)$.

Solution

The `BUILD-HEAP` algorithm calls `HEAPIFY` on internal nodes, starting from the lowest level up to the root.

Step 1: Cost of heapify at height h

The time complexity of `HEAPIFY` on a node at height h is

$$O(h).$$

Step 2: Total cost over all nodes

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

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

Therefore, the total cost of `BUILD-HEAP` is

$$\sum_{h=0}^{\log n} \frac{n}{2^{h+1}} \cdot O(h).$$

Step 3: Evaluate the summation

The above summation converges to a linear function of n, since

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

Thus,

$$\sum_{h=0}^{\log n} \frac{n}{2^{h+1}} \cdot O(h) = O(n).$$

Conclusion

Hence, the time complexity of the `BUILD-HEAP` algorithm is

$$\boxed{O(n)}.$$

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 the product of a lower triangular matrix L and an upper triangular matrix U, such that

$$A = LU.$$

This factorization is obtained using Gaussian Elimination.

Step 1: Start with the matrix A

Consider a square matrix

$$A = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{bmatrix}.$$

Step 2: Apply Gaussian Elimination

Using elementary row operations, eliminate the entries below the main diagonal to convert A into an upper triangular matrix U.

For each column k, the multiplier used to eliminate the entry a_{ik} is

$$m_{ik} = \frac{a_{ik}}{a_{kk}}, \quad i > k.$$

Step 3: Construct the matrix L

The multipliers m_{ik} used during elimination are stored in the lower triangular matrix L, while the diagonal entries of L are set to 1:

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

Step 4: Obtain the matrix U

After completing Gaussian elimination, the resulting upper triangular matrix is denoted by U:

$$U = \begin{bmatrix} u_{11} & u_{12} & \cdots & u_{1n} \\ 0 & u_{22} & \cdots & u_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & u_{nn} \end{bmatrix}.$$

Step 5: Final decomposition

Thus, the original matrix A can be expressed as

$$\boxed{A = LU}.$$

Conclusion

LU decomposition uses Gaussian elimination to transform a matrix into an upper triangular form while recording the elimination multipliers in a lower triangular matrix. This decomposition is useful for efficiently solving systems of linear equations.

Example: LU Decomposition

Let

$$A = \begin{bmatrix} 1 & 2 & 3 \\ 2 & 5 & 8 \\ 4 & 6 & 8 \end{bmatrix}.$$

Step 1: Eliminate entries below a_{11}

Pivot element: $a_{11} = 1$

Multipliers:

$$m_{21} = \frac{2}{1} = 2, \qquad m_{31} = \frac{4}{1} = 4.$$

Row operations:

$$R_2 \leftarrow R_2 - 2R_1, \qquad R_3 \leftarrow R_3 - 4R_1.$$

This gives

$$\begin{bmatrix} 1 & 2 & 3 \\ 0 & 1 & 2 \\ 0 & -2 & -4 \end{bmatrix}.$$

Step 2: Eliminate entry below a_{22}

Pivot element: $a_{22} = 1$
Multiplier:

$$m_{32} = \frac{-2}{1} = -2.$$

Row operation:

$$R_3 \leftarrow R_3 + 2R_2.$$

This gives the upper triangular matrix

$$U = \begin{bmatrix} 1 & 2 & 3 \\ 0 & 1 & 2 \\ 0 & 0 & 0 \end{bmatrix}.$$

Step 3: Construct the lower triangular matrix

$$L = \begin{bmatrix} 1 & 0 & 0 \\ 2 & 1 & 0 \\ 4 & -2 & 1 \end{bmatrix}.$$

Step 4: Verify that $A = LU$

$$LU = \begin{bmatrix} 1 & 0 & 0 \\ 2 & 1 & 0 \\ 4 & -2 & 1 \end{bmatrix} \begin{bmatrix} 1 & 2 & 3 \\ 0 & 1 & 2 \\ 0 & 0 & 0 \end{bmatrix} = \begin{bmatrix} 1 & 2 & 3 \\ 2 & 5 & 8 \\ 4 & 6 & 8 \end{bmatrix}.$$

Hence,

$$\boxed{A = LU}.$$

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

Given

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

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

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

Question 6

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

Solution

Let the matrix A be partitioned as

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

where B is a square and non-singular matrix.

Step 1: Define the Schur complement

The Schur complement of B in A is defined as

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

Step 2: Use determinant property

The determinant of a block matrix with B invertible satisfies

$$\det(A) = \det(B)\,\det(E - DB^{-1}C).$$

That is,

$$\det(A) = \det(B)\,\det(S).$$

Step 3: Use non-singularity of A

Since A is non-singular,

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

Also, since B is non-singular,

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

Step 4: Conclude non-singularity of the Schur complement
From

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

and since both $\det(A)$ and $\det(B)$ are non-zero, it follows that

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

Conclusion
Since the determinant of the Schur complement S is non-zero, S is non-singular. Hence, if matrix A is non-singular, then its Schur complement is also 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 $A \in \mathbb{R}^{n\times n}$ be a symmetric positive-definite matrix. By definition,

$$x^T Ax > 0 \quad \text{for all } x \neq 0.$$

A fundamental property of positive-definite matrices is that all their leading principal minors are positive. That is,

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

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

During LU decomposition using Gaussian elimination, the pivot at step k is the diagonal element u_{kk} of the upper triangular matrix U. The pivot u_{kk} is non-zero if and only if the leading principal submatrix A_k is non-singular.

Since $\det(A_k) > 0$ for all k, it follows that

$$u_{kk} \neq 0 \quad \text{for all } k.$$

Therefore, no division by zero occurs at any step of the elimination process, and pivoting is not required.

Conclusion
Because all pivots are guaranteed to be non-zero, every positive-definite matrix admits an 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

An augmenting path is a path from the source vertex S to the sink vertex T along which the flow can be increased.

Breadth First Search (BFS) is preferred over Depth First Search (DFS) for finding augmenting paths. BFS explores the graph level by level and always finds the shortest augmenting path in terms of the number of edges.

Using shorter augmenting paths reduces the total number of augmentations required to reach the maximum flow, thereby improving the efficiency of the algorithm. This idea is used in the Edmonds–Karp algorithm, which guarantees polynomial-time complexity.

Example

Consider the following flow network:

$$S \to A \to C \to T$$

$$S \to B \to T$$

This can be visualized as:

$$\begin{tikzcd}
& S \arrow[dl] \arrow[dr] & \\
A \arrow[d] & & B \arrow[d] \\
C \arrow[dr] & & T \\
& T &
\end{tikzcd}$$

Using DFS

Depth First Search starts from S and explores as deep as possible before backtracking. DFS may first find the path

$$S \to A \to C \to T.$$

This path contains more edges. Augmenting flow along such longer paths may lead to inefficient performance and an increased number of iterations.

Using BFS

Breadth First Search explores the graph level by level. From S, BFS examines all vertices at distance one and quickly finds the shorter path

$$S \to B \to T.$$

Since this path has fewer edges, it allows faster augmentation and reduces the total number of augmenting steps.

Conclusion

This example clearly shows that BFS finds shorter augmenting paths compared to DFS. Therefore, BFS is preferred over DFS for finding augmenting paths in flow networks.

Question 9

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

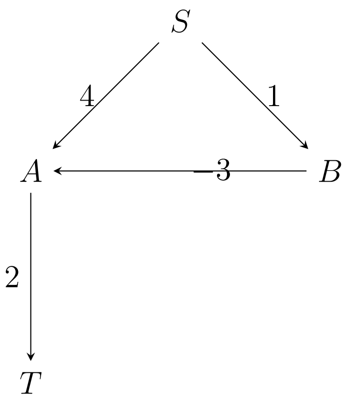
Solution

Dijkstra's algorithm is based on the assumption that once the shortest distance to a vertex is determined, that distance will never change. This assumption holds only when all edge weights in the graph are non-negative.

If a graph contains a negative edge weight, a shorter path to a vertex may be discovered later through another vertex. In such cases, Dijkstra's algorithm may finalize a vertex too early and fail to update its distance, leading to incorrect results.

Example

Consider the following directed weighted graph:



Explanation using the graph

Starting from source vertex S:

- The direct path $S \to A$ has cost 4.
- The path $S \to B$ has cost 1.

Dijkstra's algorithm will first select vertex B (distance 1) and then consider the edge $B \to A$ with weight -3. This gives a new distance to A:

$$1 + (-3) = -2.$$

However, if vertex A had already been finalized earlier with distance 4, Dijkstra's algorithm would not update it to -2. Thus, the algorithm fails to compute the correct shortest path.

Conclusion

Because negative edge weights can produce shorter paths after a vertex has been finalized, Dijkstra's algorithm does not work correctly on graphs with negative edge weights. Hence, Dijkstra's algorithm cannot be applied to such graphs.

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 a graph G. The symmetric difference of M_1 and M_2 is defined as

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

Step 1: Degree of vertices

Since M_1 and M_2 are matchings, each vertex in G is incident to at most one edge from M_1 and at most one edge from M_2. Hence, in the graph formed by $M_1 \oplus M_2$, every vertex has degree at most 2.

Step 2: Structure of connected components

In any graph where every vertex has degree at most 2, each connected component must be either:

- a path (with exactly two vertices of degree 1), or
- a cycle (with all vertices of degree 2).

Step 3: Parity of cycles

Along any cycle in $M_1 \oplus M_2$, edges must alternate between M_1 and M_2. Therefore, the number of edges in such a cycle must be even.

Conclusion

Every connected component of the symmetric difference of two matchings is either a path or an even-length cycle.

Example

Consider a graph with vertices $\{v_1, v_2, v_3, v_4\}$.

Let

$$M_1 = \{(v_1, v_2), (v_3, v_4)\}, \quad M_2 = \{(v_2, v_3), (v_4, v_1)\}.$$

The symmetric difference is

$$M_1 \oplus M_2 = \{(v_1, v_2), (v_2, v_3), (v_3, v_4), (v_4, v_1)\},$$

which forms the cycle

$$v_1 \to v_2 \to v_3 \to v_4 \to v_1.$$

This cycle has even length, confirming the result.

Question 11

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

Solution

The class **Co-NP** consists of all decision problems whose *complements* belong to the class NP.

Formally, a language L is in Co-NP if

$$\overline{L} \in \text{NP},$$

where $\overline{L}$ denotes the complement of L.

This means that for problems in Co-NP, whenever the correct answer is **NO**, there exists a certificate that can be verified in polynomial time.

Type of problems in Co-NP

Problems belonging to Co-NP have the following characteristics:

- A **NO** answer can be verified efficiently.
- Given a suitable proof or certificate, it can be checked in polynomial time that the instance does *not* satisfy the required property.

Example

The problem UNSAT (checking whether a Boolean formula is unsatisfiable) belongs to Co-NP. If a Boolean formula is unsatisfiable, a certificate can be used to verify this fact in polynomial time.

Since SAT is in NP, its complement UNSAT is in Co-NP.

Conclusion

Co-NP consists of decision problems for which negative instances admit polynomial-time verifiable proofs. It is widely believed that

$$\text{NP} \neq \text{Co-NP},$$

although this has not yet been proven.

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

A Boolean circuit consists of input variables, logic gates (AND, OR, NOT), and a single output gate. The circuit can be represented as a directed acyclic graph (DAG), where vertices correspond to gates or inputs, and edges represent connections between them.

To verify that the output of the circuit evaluates to `true`, a Depth First Search (DFS) can be performed starting from the output gate.

During DFS, the value of each gate is computed only after all its input gates have been visited. Since each gate is visited once and each edge is explored once, the total verification time is polynomial in the size of the circuit.

Example

Consider a Boolean circuit defined by

$$(A \wedge B) \vee C.$$

Suppose the input assignment is

$$A = 1, \quad B = 1, \quad C = 0.$$

Using DFS:

- The DFS starts from the output OR gate.
- It first visits the AND gate and evaluates $A \wedge B = 1$.
- It then visits input C and reads its value as 0.
- The OR gate computes $1 \vee 0 = 1$.

Since the final value at the output gate is 1, the circuit evaluates to `true`.

Conclusion

Because DFS visits each gate and wire exactly once and computes gate values locally, the correctness of a Boolean circuit output can be verified in polynomial time using DFS.

Question 13

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

Solution

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

The 3-SAT problem is a special case of the Boolean satisfiability problem (SAT), where the Boolean formula is expressed in conjunctive normal form (CNF) and each clause contains exactly three literals.

It is known that SAT is NP-Complete. Furthermore, SAT can be reduced to 3-SAT in polynomial time. This reduction transforms any Boolean formula in CNF into an equivalent 3-CNF formula without changing its satisfiability.

Since:

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

and SAT is NP-Complete, it follows that 3-SAT is NP-Hard.

Additionally, 3-SAT belongs to NP because, given a truth assignment, the satisfiability of the formula can be verified in polynomial time.

Conclusion

Because 3-SAT is both NP-Hard and belongs to NP, it is NP-Complete.

Question 14

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

Solution

The 2-SAT problem is **not NP-Hard**, and it can be solved in **polynomial time**.

In the 2-SAT problem, each clause of the Boolean formula contains at most two literals. Although the general SAT and 3-SAT problems are NP-Complete, this restriction significantly reduces the complexity of the problem.

Each clause of the form

$$(a \vee b)$$

can be rewritten as two logical implications:

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

Using these implications, a directed graph called the *implication graph* is constructed. A 2-SAT instance is satisfiable if and only if no variable and its negation belong to the same strongly connected component of this graph.

Strongly connected components can be found using algorithms such as Kosaraju's or Tarjan's algorithm, both of which run in linear time.

Therefore, the 2-SAT problem can be solved in polynomial time.

Conclusion

Since 2-SAT admits a polynomial-time solution, it is not NP-Hard (unless $P = NP$).