DAA

24 February 2024 07:26 PM

$O(g(n)) = \{f(n) \mid \exists c_1 \in \mathbb{R}^+, n_0 \in \mathbb{Z}^+ \text{ such that } 0 \le f(n) \le g(n) \text{ for all } n \ge n_0\}$

Example 2: Disprove $n^3 \ne O(n^2)$.

Ans: Assume that the proposition is true. Then $n^3 \le c \cdot n^2$ (for some n_o, such that, it is valid $\forall n \ge n_o$).
But for $n \ge c+1, (c+1)^3 \ge c \cdot (c+1)^2$. We arrive at a contradiction. So, the proposition is false, and $n^3 = O(n^2)$.

Example 5: Find time complexity. (Number of digits in a binary number)

```
1: count ← 1
2: while n > 1 do
3:     count ← count + 1
4:     n ← n/2
5: end while
```

Ans: For simplicity, assume $n = 2^i$. The value of n in the loop would progress as follows:

$$n, \frac{n}{2}, \frac{n}{2^2}, \dots, \frac{n}{2^{i-1}}$$

$\Omega(g(n)) = \{f(n) : \exists c \in \mathbb{R}^+ \text{ and } n_0 \in \mathbb{Z}^+ \text{ such that } 0 \le c \cdot g(n) \le f(n)\}$

2.7 Average-case Time Complexity

Definition: $\Theta(g(n)) = \{f(n) : \exists c_1 \in \mathbb{R}^+ \text{ and } \exists c_2 \in \mathbb{R}^+ \text{ and } n_0 \in \mathbb{Z}^+ \text{ such that } c_1 \cdot g(n) \le f(n) \le c_2 \cdot g(n)\}$

$f(n) = \theta(g(n))$ iff $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$

3.1 o-Notation

Definition:

$o(g(n)) = \{f(n) \mid \exists n_0 \in \mathbb{Z}^+, \text{ and } \forall c \in \mathbb{R}^+ \text{such that } 0 \le f(n) < c.g(n) \ \forall n \ge n_0\}$

and

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$$

3.2 ω-Notation

Definition:

$\omega(g(n)) = \{f(n) \mid \exists n_0 \in \mathbb{Z}^+, \text{ and } \forall c \in \mathbb{R}^+ \text{such that } 0 \le c.g(n) < f(n) \ \forall n \ge n_0\}$

and

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty$$

6 Master's theorem

Definition: For $a \ge 1$, $b \ge 1$ and a non-negative function $f(n)$

$$T(n) = aT(n/b) + f(n) \text{ and } T(1) = \theta(1)$$

then

$$T(n) = \theta(n^{\log_b a}) + \sum_{j=0}^{\log_b n - 1} a^j f(\frac{h}{b^j})$$

and

- If there exists a constant $\epsilon > 0$ such that $f(n) = O(n^{\log_b a - \epsilon})$, then $T(n) = \theta(n^{\log_b a})$.
- If $f(n) = \theta(n^{\log_b a})$, then $T(n) = \theta(n^{\log_b a} log\ n)$.
- If there exists a constant $\epsilon > 0$ such that $f(n) = \Omega(n^{\log_b a + \epsilon})$ and $af(n/b) \le cf(n)$ for $c < 1$, then $T(n) = \theta(f(n))$.

Case 3: When $f(n) = \Omega(n^{\log_b a + \epsilon})$ for some $\epsilon > 0$ and $af(n/b) \le cf(n)$ for $c < 1$.

Since we have $af(n/b) \le cf(n)$, using induction we can write that for j iterations would yield

$$a^j(f(\frac{n}{b^j})) \le c^j(f(n))$$

$$\begin{aligned} g(n) &= \sum_{j=0}^{\log_b n} a^j f(n/b^j) \\ &\le \sum_{j=0}^{\log_b n} c^i f(n) \\ &= f(n) \sum_{j=0}^{\log_b n} c^i \end{aligned}$$

2.4 Use-Cases

Merge sort is useful in distributed systems in which the problem is unsolvable for one memory unit. Therefore, it is distributed among multiple memory units and the solutions are recombined to obtain the final solution,

Merge sort is less useful for low-memory systems.

Since **Divide** step precedes the **Conquer** step, merge-sort is recursive.

Quick sort can be iterative as well since **Partition** step has the precedence.

3.5 Use-Cases

Quick Sort is useful in all the cases except for sorted array and reverse sorted array.Quick Sort is an in-place algorithm. However, the space complexity is O(log(n)) due to the usage of system stack.

Quick Sort is used in many inbuilt sorting algorithms like sort() , because the probability of getting worst cases are very less and also in-place as well.

Quick Sort can be implemented using both recursive approach and iterative approach.

HEAPS

1.1 Structural Property

A heap has two main properties:

- **Complete Binary Tree:** A heap is a complete binary tree. All levels are filled, except possibly the last one, which is filled from left to right.
- **Level Completeness:** If there are l levels in a heap, then levels up to $(l-1)$ are fully filled, and the l-th level is left-filled (may not be completely filled).

If there are h levels, then the total number of elements for a heap of height h will be:

$$2^0 + 2^1 + 2^2 + \ldots + 2^{h-1} + 2^h = 2^{h+1} - 1$$

- The time complexity for inserting a node into a heap is $O(\log n)$, where n is the total number of nodes. This complexity is based on the logarithmic height (h) of the heap.

- To apply heapify at a node, its right and left subtrees should follow the heap structure.

- Therefore, the time complexity to build a min/max heap is $O(n\log(n))$.

1.7.1 Advantages of Heapsort

- This is an in-place algorithm, hence is better than merge-sort, which has linear space complexity sorting
- The worst case time complexity of this algorithm is $O(n\log(n))$ which is sometimes better than quick sort which has the worst case time complexity as $O(n^2)$ when the elements of the array are reverse sorted.

Due to multiplication with a constant factor, which will determine the ranking in terms of complexity

- For practical uses, Quicksort algorithm is much faster than the Heapsort, because its inner loop can be efficiently implemented on most architectures
- Merge-sort is preferable when merging two arrays and then sorting them in a large database

Heapsort can be more efficient, if we use ternary tree or use m^{th} order tree. Because as the number of children will increase, the height of the tree will decrease.

GRAPH

3 Graph Representation

$G = (V, E)$ and $E \subseteq V \times V$
A graph G is defined by a set of vertices (V) and a set of edges (E) such that each edge is a subset of the Cartesian product of V with itself.

3.4 Explicit Graph

An explicit graph is one where all vertices and edges are given.We have a finite set of vertices and edges.

3.5 Implicit Graph

In contrast to an explicit graph, in scenarios where we only know the start node, the subsequent nodes and edges are built gradually under a certain set of rules and conditions.

Theoretically, this graph can be infinite, depending on the domain of possible states.
We cannot represent the graph using a matrix because we don't know the set of vertices beforehand. Therefore, it is suggested to use an adjacency list.

- For undirected graph maximum number of edges possible is $\binom{n}{2}$ and for directed graph it is n(n-1).
- In the case of an undirected graph, storing it in a matrix has the drawback of containing redundant data.
- In cases of storing sparse data, where the degree of nodes is significantly smaller than the number of nodes, it is desirable to use an adjacency list.
- For determining whether a vertex i is a neighbor of vertex j, the matrix representation can achieve it in O(1), whereas in the case of adjacency list representation, it takes O(branching factor).
- For determining all the neighbors of vertex i, the matrix representation can achieve it in O(n), whereas in the case of adjacency list representation, it takes O(branching factor).
- The vectorized operations that occur in our computer systems store data in matrices, enabling efficient matrix manipulations.

Complexity for BFS:

Adjacency Matrix:

- **Space Complexity:** $O(|V|^2)$ for the matrix.
- **Time Complexity:** $O(|V|^2)$ due to the need to check all vertices.

Adjacency List:

- Space Complexity: $O(|V| + |E|)$ for the list.
- Time Complexity: $O(|V|+|E|)$, where V is the number of vertices and E is the number of edges. This complexity is approximately linear when the graph is sparse, offering significant efficiency improvements. However, in the case of a dense graph, where the number of edges approaches the maximum possible $(|V|^2)$, the time complexity may exhibit characteristics closer to $O(|V|^2)$ due to the potential presence of a substantial number of edges.

Implicit Graph:

- Analogous to a binary tree, for an implicit graph, the time complexity of traversing the nodes through BFS is $O(b^d)$, where b is the branching factor and d is the depth of the graph.
  Explanation:
  In the context of an implicit graph, the time complexity for BFS is expressed as $O(b^d)$, drawing an analogy to a binary tree. This relationship arises from the fact that the traversal involves exploring all nodes up to a certain depth. The branching factor b represents the number of child nodes each node can have, and the depth d represents the maximum depth of the implicit graph. As BFS explores nodes level by level, the time complexity grows exponentially with the branching factor and the depth of the graph. Therefore, the notation $O(b^d)$ captures the computational cost associated with traversing the implicit graph through BFS.

The time complexity of DFS is different depending on the graph representation used:

- **Adjacency Matrix**: In the case of an adjacency matrix, the time complexity of DFS is $O(V^2)$, where V is the number of vertices. This is because we need to check all possible edges for each vertex.
- **Adjacency List**: When using an adjacency list, the time complexity of DFS is $O(V + E)$, where V is the number of vertices and E is the number of edges in the graph. This is because, for each vertex, we need to traverse its adjacency list, which takes $O(\text{degree}(v))$ time, where $\text{degree}(v)$ is the number of edges incident to vertex v. The sum of all degrees in the graph is $2E$, so the total time complexity is $O(V + 2E) = O(V + E)$.

In the worst-case scenario where every vertex is connected to every other vertex, $E = V^2$, and therefore $O(V+E)$ becomes $O(V^2)$.

The space complexity of DFS is $O(V + V)$ in case of explicit graph, where V is for visited array used and another V is the worst case length of recursion stack in case of skewed graph. In case of implicit graph, space complexity is $O(bd)$ where b is the branching factor and d is the depth of the graph.

Completeness in algorithms means that the algorithm will always find a solution within a reasonable amount of time if at least one solution exists.For example, DFS and BFS are two fundamental graph traversal algorithms used to explore and search for nodes in a graph. While both algorithms are widely used and effective in various scenarios, they differ in their completeness. BFS is considered a complete algorithm, on the other hand, DFS is not considered a complete algorithm.

Why DFS is Not Complete:

- **Unbounded Exploration:** In the case of implicit graphs, DFS may continue indefinitely along an unexplored branch where the destination could be located.

- **Lack of Shortest Path Guarantee:** DFS does not guarantee the discovery of the shortest path between the starting node and any other node. Due to its nature of exploring one branch deeply, it may find a longer path before discovering a shorter one.

- **Shortest Path Property:** BFS discovers the shortest path from the starting node to any reachable node. Since it explores level by level, the first occurrence of a node guarantees the shortest path to that node.

A connected graph is a graph in which there exists a path between every pair of vertices. In other words, there are no isolated vertices, and all vertices are reachable from every other vertex.

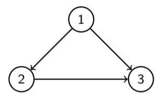
Algorithm 3 Count Connected Components

```
function CountComponents($G$)
    $visited \leftarrow$ Empty set
    $count \leftarrow 0$
    for all $v$ in $G$ do
        if $v$ is not visited then
            DFS_Visit($G, v$) {or BFS_Visit}
            $count \leftarrow count + 1$
        end if
    end for
    return $count$
end function

function DFS_Visit($G, v$)
    Mark $v$ as visited
    for all $u$ in $G.adj[v]$ do
        if $u$ is not visited then
            DFS_Visit($G, u$)
        end if
    end for
end function
```

Algorithm 4 Detect Cycle in Undirected Graph using BFS

```
function DetectCycle($G$)
$visitedEdges \leftarrow$ empty hashmap {Initialize empty hashmap to track visited edges}
$queue \leftarrow$ empty queue
for each vertex $v$ in $G$ do
    if $v$ is not visited then
        enqueue $v$ into $queue$
        mark $v$ as visited
        while $queue$ is not empty do
            $u \leftarrow$ dequeue from $queue$
            for each neighbor $w$ of $u$ do
                if $(u, w)$ is not in $visitedEdges$ then
                    add $(u, w)$ to $visitedEdges$
                    enqueue $w$ into $queue$
                end if
            end for
        end while
    end if
end for
for each edge $(u, v)$ in $E(G)$ do
    if $(u, v)$ is not in $visitedEdges$ then
        return false {Cycle detected}
    end if
end for
return true {No cycle detected} =0
```

The above algorithm would not work in case of directed graphs. Consider the example given below



Suppose we start BFS from node 1, then after the completion, edges $1 \longrightarrow 2$ and $1 \longrightarrow 3$ will be marked visited and the BFS will end. The edge $2 \longrightarrow 3$ will be unvisited and the algorithm will say cycle is present, but as we can see, there is no cycle.

5 DFS Numbering

DFS numbering assigns a unique number to each vertex of a graph during a depth-first search traversal. The numbering reflects the order in which vertices are discovered and processed.

Algorithm 5 DFS with Numbering

```
Input: Graph $G$ with vertices $V$ and edges $E$, starting vertex $s$
Output: Depth-first traversal of $G$ starting from $s$ with vertex numbering
Procedure: DFS_Numbering($G, s, count$)
Initialize a tuple $t$ $(\infty, \infty)$
Mark $s$ as visited
$t \leftarrow (count, \infty)$
$count \leftarrow count + 1$
for each vertex $v$ in $G.adj[s]$ do
    if $v$ is not visited then
        DFS_Numbering($G, v, count$)
    end if
end for
$t \leftarrow (t.first, count)$
$count \leftarrow count + 1$ =0
```

6 Articulation Points

In graph theory, an articulation point (or cut vertex) is a vertex in a graph whose removal would disconnect the graph. Formally, a vertex v is an articulation point if and only if its removal increases the number of connected components in the graph.

1 Topological Sorting

Topological sorting is an algorithm for linear ordering of vertices in a Directed Acyclic Graph (DAG) such that for every directed edge u-v, vertex u comes before v in the ordering.

A Directed Acyclic Graph (DAG) is a data structure that has allowing features:

- Directed: The edges of the graph have a direction, indicated by arrows. This implies a one-way dependency or flow between nodes.
- Acyclic: The graph does not contain any cycles. This means you cannot start at a node, follow the direction of the edges, and end up back at the same node.

In graph we can represent directed edges(i,j) where course i is a prerequisite for course i.

Note: The above example is a partial ordering(reflexive,anti-symmetric,transitive) not a total ordering as directed edge from i to j means no directed edge from j to i.

Question: Prove that in a DAG a situation with no node with indegree=0 is not possible and prove that topological sorting exist for DAG.
Hint: use contradiction

1.2 Algorithm of Topological Sorting

Steps:

1. push all vertex with indegree=0 in a queue.
2. remove a vertex and reduce the indegree of all its neighbour vertices by 1. If the indegree of the vertices become 0 push into queue.

Algorithm:

```
Topological-Sort(G)
for each vertex u in G.V
    for each vertex v in G.Adj[u]
        in_degree[u]=in_degree[u]+1
for each vertex u in G.V
    if in_degree[u]==0

         Enqueue(Q,u)
 while Q is not empty
     u=Dequeue(Q)
     for each v in G.Adj[u]
         in_degree[v]--
         if in_degree[v]==0
             Enqueue(v)
```

- In a Directed Acyclic Graph (DAG), a topological sort is required for finding the longest path because the longest path can be determined by traversing the nodes in a specific order.

1.4 Time Complexity Analysis of Topological Sorting

1. Push the vertex with indegree 0
   - for Adj Matrix– $O(V^2)$
   - for Adj List– $O(E)$
2. Remove the vertex with indegree 0 and visit its neighbour and reduce its indegree by 1
   - for Adj Matrix– $O(V^2)$
   - for Adj List– $O(E + V)$
3. Longest path of the graph
   - for Adj Matrix– $O(V^2)$
   - for Adj List– $O(E + V)$

SHORTEST PATH PROBLEM

Given a weighted and directed graph $G = (V, E)$ and $W : E \longrightarrow R$ (R represents Real Numbers), the cost of a path $P_{(v_0,v_k)}$ from vertex v_0 to v_k is defined as

$$W(P_{(v_0,v_k)}) = \sum_{i=0}^{k-1} W(v_i, v_{i+1})$$

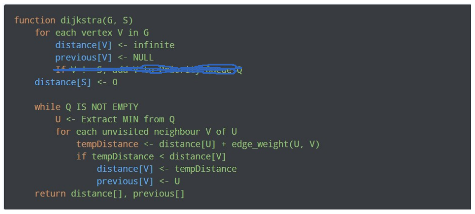
Shortest Path $S(v_0, v_k)$ is defined as

$$S(v_0, v_k) = \begin{cases} \min(W(P_{(v_0,v_k)})) & \text{if } \exists \text{ a path between } v_0 \text{ and } v_k \\ \infty & \text{otherwise} \end{cases}$$

3 Dijkstra's Algorithm

Below is the implementation of the Dijkstra's Algorithm:

```
  Input: Graph $G$, starting vertex $s$
  Output: Shortest distances from $s$ to all other vertices
1 Make array for storing distances of each vertex from the starting vertex
  Make array for storing the parent of each of the vertices
  for each vertex $v \in G.V$ do
2     $v.d \leftarrow$ distance between $s$ (source vertex) and $v$
      $v.d \leftarrow \infty$
      v.pie $\leftarrow$ null (parent)
3 $s.d \leftarrow 0$ Distance from source vertex to itself
  Priority Queue $Q \leftarrow G.V$
  $s$.pie $\leftarrow \phi$
  while $Q$ is not empty do
4     $u \leftarrow Q$.extractMin()
      $s \leftarrow s \cup \{u\}$
      Mark $u$ as visited
5     for each neighbor $v$ of $u$ do
6         if $v$ does not belong to $s$ then
7             if $v.d > u.d + w(u,v)$ (edge weight between vertices $u$ and $v$) then
8                 $v.d \leftarrow u.d + w(u,v)$
                  $v$.pie $\leftarrow u$
9 return Shortest distances array
```



```
function dijkstra(G, S)
    for each vertex V in G
        distance[V] <- infinite
        previous[V] <- NULL
        ~~If V != S, add V to Priority Queue Q~~
    distance[S] <- 0

    while Q IS NOT EMPTY
        U <- Extract MIN from Q
        for each unvisited neighbour V of U
            tempDistance <- distance[U] + edge_weight(U, V)
            if tempDistance < distance[V]
                distance[V] <- tempDistance
                previous[V] <- U
    return distance[], previous[]
```

Using adjacency matrix :

Initialization: Initializing the distances and priority queue takes $O(|V|)$ time
Main Loop:

1. **ExtractMin Operation**:Extracting the minimum distance vertex from the priority queue takes $O(log(|V|))$ time
2. **Neighbour Exploration**:For each neighbor of the current vertex, the algorithm explores the neighbors and updates their tentative distances. In the worst case, each edge is considered once, resulting in $O(|V|)$ time for the loop.

The main loop iterates V times (once for each vertex), and in each iteration, there is a $O(log(|V|))$ operation (extracting the minimum distance vertex) and a $O(V)$ operation (neighbor exploration).

Total Time Complexity: The total time complexity of the main loop is $O(V.(log(|V|) + V))$. and the time complexity can be approximated as $O(|V|^2)$

Using adjacency list:

We can observe that the statements in the inner loop are executed $O(V+E)$ times (similar to BFS). The inner loop has `extractMin()` operation which takes $O(\log V)$ time. So overall time complexity is $O((E + V) \cdot \log V) = O(E \log V)$.

Note that the above code uses Binary Heap for Priority Queue implementation. Time complexity can be reduced to $O(E + V \log V)$ using Fibonacci Heap. The reason is that Fibonacci Heap takes $O(1)$ time for `extractMin` operation while Binary Heap takes $O(log(|V|))$ time.

In summary, Dijkstra's algorithm using an adjacency list representation is efficient with a time complexity that scales well, particularly in graphs where the number of edges is significantly less than the square of the number of vertices.

3.3.2 Space Complexity Analysis:

$|V|$: total number of vertices

Distances Array: The distances array stores the current shortest distances from the source vertex to all other vertices. It requires $O(|V|)$ space.

Priority Queue: The priority queue is used to efficiently extract the minimum distance node during each iteration. Depending on the implementation, the space complexity of the priority queue can vary. A binary heap-based priority queue requires $O(log(|V|))$ space.

Total Space Complexity: Combining the space requirements of these data structures, the overall space complexity of Dijkstra's algorithm with a binary heap-based priority queue is $O(|V|)$

A loop invariant is a property that holds true before and after each iteration of a loop.

3.2 How to check the correctness of Dijkstra's Algorithm ?

1. **Initialization:** Before the loop starts, the loop invariant holds true. In the context of Dijkstra's algorithm, this could be the correct initialization of distances and predecessors.

2. **Maintenance:** Assuming the loop invariant is true before an iteration, the algorithm maintains it during the iteration. In Dijkstra's algorithm, this corresponds to the correct relaxation step, updating distances and predecessors.

3. **Termination:** When the loop terminates, the loop invariant guarantees the correctness of the algorithm's output. In Dijkstra's algorithm, the loop terminates when all vertices are included in the shortest path tree.

1 Bellmann-Ford Algorithm

The Bellman-Ford algorithm is preferred over Dijkstras algorithm in scenarios where negative edge weights or cycles exist in the graph. Bellman-Ford can handle negative weights and detect negative cycles, features that Dijkstra lacks due to its reliance on a greedy choice, which is not always suitable in the presence of negativity.

Given a weighted, directed graph $G = (V, E)$ with a source vertex s and weight function $w : E \to \mathbb{R}$, the Bellman-Ford algorithm returns a boolean value indicating the presence of a reachable negative-weight cycle from the source. If such a cycle exists, the algorithm signals that no solution is possible. Conversely, if there is no negative-weight cycle, the algorithm computes the shortest paths and their corresponding weights.

1.1 Working Principles

It works on two principles:

1. Shortest path from one node to another doesn't contain any cycle.

2. If $s \to t$ represents shortest path from s to t then $s \to v_i$ for any v_i in path from s to t also produces shortest path.

Proof: Let p, p_1 and p_2 be shortest path from s to t, subpath of p from s to v_i and subpath of p from v_i to t respectively.

$\therefore p = p_1 + p_2$.

Let p_1' be a shortest path from s to v_i and p' be a path from s to t through p_1' , therefore, p'=p_1' + p_2. But p is shortest path from s to t, therefore p$\leq p'$,

$\implies p_1 + p_2 \leq p'_1 + p_2$

$\implies p_1 \leq p'_1$

But p_1' is shortest path from s to v_i, therefore, $p_1 = p_1'$.

Hence proved.

Algorithm 1 Bellman-Ford

```
1: Input: Graph $G$ with vertices $V$ and edges $E$, source vertex $s$ and weight function $w$
2: Output: Shortest path from given vertex to all other vertices.
3: Procedure: BELLMAN-FORD$(G, w, s)$
4: for each vertex $v \in$ G.V
5:     $v.d = \infty$
6:     $v.\pi$=NIL      Dist source =0
7: for i = 1 to $|G.V|$ - 1
8:     for each edge $(u, v) \in G.E$
9:         if $v.d > u.d + w(u, v)$
10:            $v.d = u.d + w(u, v)$
11:            $v.\pi = u$
12: // To check if -ve cost cycle exists
13: for each edge $(u, v) \in G.E$
14:     if $v.d > u.d + w(u, v)$
15:         return FALSE
16: return TRUE
    =0
```

The **Bellman-Ford** algorithm has a time complexity of **O(V*E)**, where V is the number of vertices and E is the number of edges in the graph. In the worst-case scenario, the algorithm needs to iterate through all edges for each vertex, resulting in this time complexity. The space complexity of the Bellman-Ford algorithm is O(V), where V is the number of vertices in the graph. This space complexity is mainly due to storing the distances from the source vertex to all other vertices in the graph.

Adjacency List

Operation	Time Complexity	Space Complexity
Initialization	O(V)	O(V)
Relaxation	O(V*E)	O(1)
Overall	O(V*E)	O(V)

Adjacency Matrix

Operation	Time Complexity	Space Complexity
Initialization	O(V)	O(V)
Relaxation	O(V^3)	O(1)
Overall	O(V^3)	O(V)

Let $\delta(u, v)$ denote the shortest path weight from u to v.

Lemma 1:

Consider a weighted, directed graph $G = (V, E)$ with a source vertex s and weight function $w : E \to \mathbb{R}$. Assuming G contains no negative-weight cycles reachable from s, after the execution of the **for** loop (lines 7 to 11) in the BELLMAN-FORD algorithm for $|V| - 1$ iterations, we have $v.d = \delta(s, v)$ for all vertices v that are reachable from s.

Proof: We prove the lemma by appealing to the path-relaxation property. Consider any vertex v that is reachable from s, and let $p = \{v_0, v_1, \ldots, v_k\}$, where $v_0 = s$ and $v_k = v$, be any shortest path from s to v. Because shortest paths are simple, p has at most $|V| - 1$ edges, and so $k \leq |V| - 1$. Each of the $|V| - 1$ iterations of the **for** loop of lines 7 to 11 relaxes all $|E|$ edges. Among the edges relaxed in the ith iteration, for $i = 1, 2, \ldots, k$, is (v_{i-1}, v_i). By the path-relaxation property, therefore,

$$v.d = v_k.d = \delta(s, v_k) = \delta(s, v).$$

Lemma 2:

For a weighted, directed graph $G = (V, E)$ with a source vertex s and weight function $w : E \to \mathbb{R}$, the BELLMAN-FORD algorithm terminates with $v.d < \infty$ for each vertex $v \in V$ if and only if there exists a path from s to v.

1.5 Correctness of the Bellman-Ford algorithm

Let **BELLMAN-FORD** be run on a weighted, directed graph $G = (V, E)$ with source s and weight function $w : E \to \mathbb{R}$. If G contains no negative-weight cycles that are reachable from s, then the algorithm returns TRUE, where $v.d = \delta(s, v)$ for all vertices $v \in V$, and the predecessor sub-graph G_π is a shortest-paths tree rooted at s. If G contains a negative-weight cycle reachable from s, then the algorithm returns FALSE.

Proof: Suppose that graph G contains no negative-weight cycles that are reachable from the source s. We first prove the claim that at termination, $v.d = \delta(s, v)$ for all vertices $v \in V$. If vertex v is reachable from s, then Lemma 1 proves this claim. If v is not reachable from s, then the claim follows from the no-path property. Thus, the claim is proven. The predecessor-subgraph property, along with the claim, implies that G_π is a shortest-paths tree. Now we use the claim to show that BELLMAN-FORD returns TRUE. At termination, for all edges $(u, v) \in E$ we have

$$v.d = \delta(s, v) \leq \delta(s, u) + w(u, v) = u.d + w(u, v),$$

and so none of the tests in line 14 causes BELLMAN-FORD to return FALSE. Therefore, it returns TRUE.

Now, suppose that graph G contains a negative-weight cycle reachable from the source s. Let this cycle be $c = \{v_0, v_1, \ldots, v_k\}$, where $v_0 = v_k$, in which case we have

$$\sum_{i=1}^{k} w(v_{i-1}, v_i) < 0 \quad — (1)$$

Assume for the purpose of contradiction that the Bellman-Ford algorithm returns TRUE. Thus, $v_i.d \leq v_{i-1}.d + w(v_{i-1}, v_i)$ for $i = 1, 2, \ldots, k$. Summing the inequalities around cycle c gives

$$\sum_{i=1}^{k} v_i.d \leq \sum_{i=1}^{k} (v_{i-1}.d + w(v_{i-1}, v_i)) = \sum_{i=1}^{k} v_{i-1}.d + \sum_{i=1}^{k} w(v_{i-1}, v_i)$$

Since $v_0 = v_k$, each vertex in c appears exactly once in each of the summations. $\sum_{i=1}^{k} v_{i-1}.d = \sum_{i=1}^{k} v_i.d$.

Moreover, by Lemma 2, $v_{i-1}.d$ is finite for $i = 1, 2, \ldots, k$. Thus, $0 \leq \sum_{i=1}^{k} w(v_{i-1}, v_i)$, which contradicts inequality (1). We conclude that the Bellman-Ford algorithm returns TRUE if graph G contains no negative-weight cycles reachable from the source, and FALSE otherwise.

Decision Tree Model:

For searching, each node in the decision tree represents a comparison between elements. Each edge leaving a node corresponds to the possible outcomes of that comparison. Leaves of the tree represent the possible search outcomes (found or not found).

Table 1: **Decision tree components(Analogy)**

Decision Tree	Algorithm
Internal node	Comparison
Leaf node	Solution
Path from root to leaf	One execution of the algorithm
Path length	Running time
Height of tree	Worst case Running time

Leaf Nodes:

If there are n elements in the data set, there are atleast n possible outcomes for searching (found at position 1, found at position 2, ..., not found or found in between elements). The height of the decision tree corresponds to the number of comparisons required.

Lower Bound Analysis:

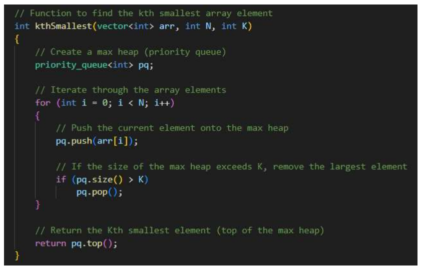
For any binary tree with height h, there are at most 2^h leaves. In our case, we want to find the minimum height h such that $2^h \geq n+1$. Solving for h, we get $h \geq \log_2(n+1) - 1$. Therefore, the lower bound for the number of comparisons in a search algorithm is $\Omega(\log n)$.

Leaf Nodes:

There are atmost $n!$ possible permutations of the n elements, and each leaf corresponds to one possible permutation. Few of those $n!$ permutations are not possible due to the above mentioned constraints in the decision tree.

The number of leaves in the decision tree must be at least $n!$ (factorial). The height of the decision tree is then at least $\log_2(n!)$, and by Stirling's approximation, it's approximately $\Omega(n \log n)$.

Prove : $\log(n!) = \Omega(n \log n)$



```
// Function to find the kth smallest array element
int kthSmallest(vector<int> arr, int N, int K)
{
    // Create a max heap (priority queue)
    priority_queue<int> pq;

    // Iterate through the array elements
    for (int i = 0; i < N; i++)
    {
        // Push the current element onto the max heap
        pq.push(arr[i]);

        // If the size of the max heap exceeds K, remove the largest element
        if (pq.size() > K)
            pq.pop();
    }

    // Return the Kth smallest element (top of the max heap)
    return pq.top();
}
```

1 Floyd-Warshall Algorithm

This Algorithm is used to find the shortest paths between all pair of nodes in a weighted graph. While we can write Bellmann Ford Algorithm for every node in order to obtain the shortest distance between every pair,it has an $O(n^4)$ time complexity and requires numerous repetitions.

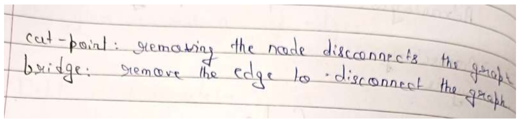
Floyd Warshall Algorithm is preferred in order to minimize repetitions and optimize time complexity. It is highly efficient and can handle graphs with both positive and negative edges.

The Floyd-Warshall algorithm assumes that the optimal path between any two vertices does not contain any negative cycles. This assumption is fundamental to the correctness of the algorithm. If there exists a negative cycle, then it is

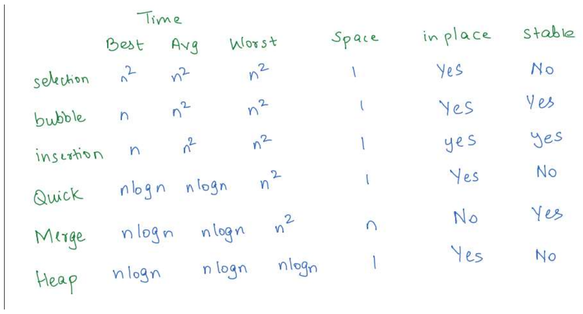
The time complexity of the Floyd-Warshall algorithm is $O(n^3)$ because for every vertex taken as an intermediate, we traverse the entire matrix. Thus, the time complexity becomes $n \times n^2 = O(n^3)$. And we maintain an adjacency matrix of $n \times n$, so space complexity is $O(n^2)$.

Listing 1: Pseudocode for Floyd-Warshall Algorithm

```
for each vertex v
    for each vertex u
        dist[v][u] gets Infinity
for each vertex v
    dist[v][v] \gets 0
for each edge (u, v) with weight w
    dist[u][v] gets w
for k from 1 to |V|
    for i from 1 to |V|
        for j from 1 to |V|
            dist[i][j] gets min(dist[i][j], dist[i][k] + dist[k][j])
```

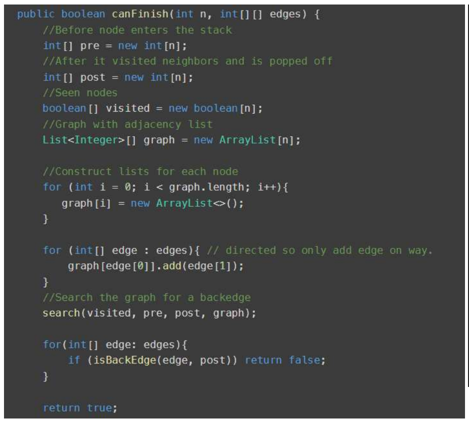


cut-point: removing the node disconnects the graph
bridge: remove the edge to disconnect the graph

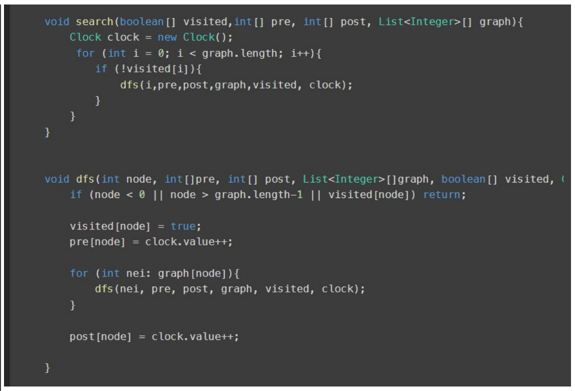


	Time					
	Best	Avg	Worst	Space	in place	stable
selection	n^2	n^2	n^2	1	Yes	No
bubble	n	n^2	n^2	1	Yes	Yes
insertion	n	n^2	n^2	1	yes	yes
Quick	nlogn	nlogn	n^2	1	Yes	No
Merge	nlogn	nlogn	n^2	n	No	Yes
Heap	nlogn	nlogn	nlogn	1	Yes	No

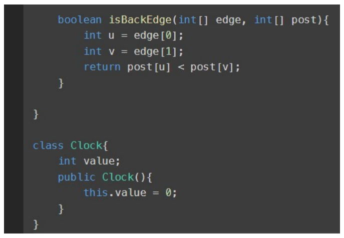
Checking cycle through DFS Numbering



```java
public boolean canFinish(int n, int[][] edges) {
    //Before node enters the stack
    int[] pre = new int[n];
    //After it visited neighbors and is popped off
    int[] post = new int[n];
    //Seen nodes
    boolean[] visited = new boolean[n];
    //Graph with adjacency list
    List<Integer>[] graph = new ArrayList[n];

    //Construct lists for each node
    for (int i = 0; i < graph.length; i++){
       graph[i] = new ArrayList<>();
    }

    for (int[] edge : edges){ // directed so only add edge on way.
        graph[edge[0]].add(edge[1]);
    }
    //Search the graph for a backedge
    search(visited, pre, post, graph);

    for(int[] edge: edges){
        if (isBackEdge(edge, post)) return false;
    }

    return true;
```

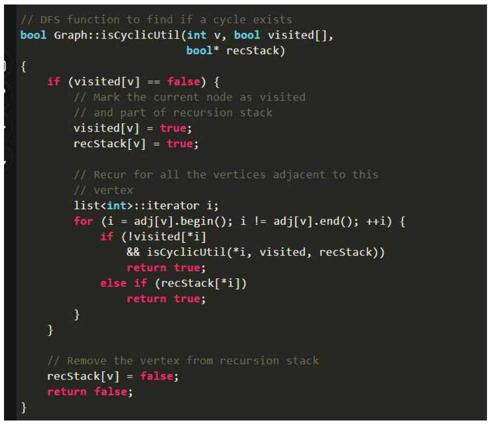


```java
void search(boolean[] visited,int[] pre, int[] post, List<Integer>[] graph){
    Clock clock = new Clock();
     for (int i = 0; i < graph.length; i++){
        if (!visited[i]){
            dfs(i,pre,post,graph,visited, clock);
        }
    }
}


void dfs(int node, int[]pre, int[] post, List<Integer>[]graph, boolean[] visited, (
    if (node < 0 || node > graph.length-1 || visited[node]) return;

    visited[node] = true;
    pre[node] = clock.value++;

    for (int nei: graph[node]){
        dfs(nei, pre, post, graph, visited, clock);
    }

    post[node] = clock.value++;

}
```

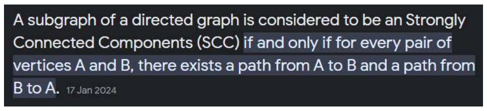


```java
    boolean isBackEdge(int[] edge, int[] post){
        int u = edge[0];
        int v = edge[1];
        return post[u] < post[v];
    }

}

class Clock{
    int value;
    public Clock(){
        this.value = 0;
    }
}
```

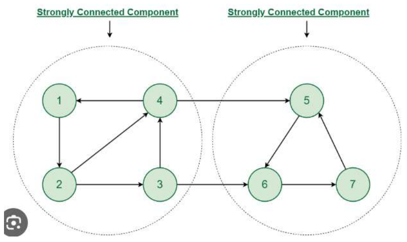
Cycle in dag

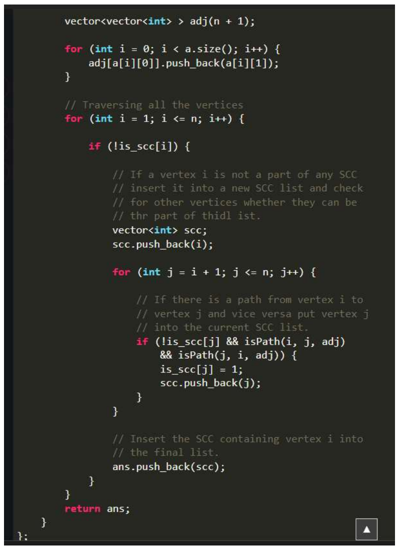


```cpp
// DFS function to find if a cycle exists
bool Graph::isCyclicUtil(int v, bool visited[],
                         bool* recStack)
{
    if (visited[v] == false) {
        // Mark the current node as visited
        // and part of recursion stack
        visited[v] = true;
        recStack[v] = true;

        // Recur for all the vertices adjacent to this
        // vertex
        list<int>::iterator i;
        for (i = adj[v].begin(); i != adj[v].end(); ++i) {
            if (!visited[*i]
                && isCyclicUtil(*i, visited, recStack))
                return true;
            else if (recStack[*i])
                return true;
        }
    }

    // Remove the vertex from recursion stack
    recStack[v] = false;
    return false;
}
```

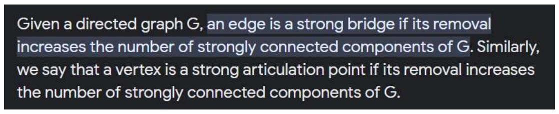


A subgraph of a directed graph is considered to be an Strongly Connected Components (SCC) if and only if for every pair of vertices A and B, there exists a path from A to B and a path from B to A. 17 Jan 2024





```cpp
        vector<vector<int> > adj(n + 1);

        for (int i = 0; i < a.size(); i++) {
            adj[a[i][0]].push_back(a[i][1]);
        }

        // Traversing all the vertices
        for (int i = 1; i <= n; i++) {

            if (!is_scc[i]) {

                // If a vertex i is not a part of any SCC
                // insert it into a new SCC list and check
                // for other vertices whether they can be
                // thr part of thidl ist.
                vector<int> scc;
                scc.push_back(i);

                for (int j = i + 1; j <= n; j++) {

                    // If there is a path from vertex i to
                    // vertex j and vice versa put vertex j
                    // into the current SCC list.
                    if (!is_scc[j] && isPath(i, j, adj)
                        && isPath(j, i, adj)) {
                        is_scc[j] = 1;
                        scc.push_back(j);
                    }
                }

                // Insert the SCC containing vertex i into
                // the final list.
                ans.push_back(scc);
            }
        }
        return ans;
    }
};
```



Given a directed graph G, an edge is a strong bridge if its removal increases the number of strongly connected components of G. Similarly, we say that a vertex is a strong articulation point if its removal increases the number of strongly connected components of G.