CS 5800: Algorithms SEC 05 Summer Full 2024
Synthesis2

Payal Chavan

07/10/2024

Chapter 1

Graph Theory

1.1 Introduction

Graphs are essential data structures that are frequently used to represent object relationships in the real world. Everywhere you look, you can see graphs: in social networks, blockchains, Google Maps, the World Wide Web, and neural networks. In computer science, effectively handling complicated problems requires an understanding of how to represent graphs efficiently.

The types, terminology, operations, representation, and applications of graphs in data structures will all be covered in detail in this chapter.

What is a Graph?

A graph is a fundamental concept in computer science, serving as an abstract representation of interconnected objects.
It consists of two primary components: vertices, which represent the objects, and edges, which indicate the connections between them.
Formally, a graph is defined as a pair of sets (V, E), where V represents the vertices and E represents the edges connecting those vertices.

A map is a common example of a graph, which is a non-linear data structure. In a map, various connections exist between cities, such as roads, railway lines, and aerial networks. We can consider the graph as representing the interconnections of cities through roads.
Another such example is Facebook. Facebook employs a graph data structure that consists of entities and their relationships. In this structure, each user, photo, post, page, and place is represented as a vertex (node). The edges connecting these vertices (nodes) signify various relationships, such as friendships, ownerships, and tags. For instance, when a user posts a photo or comments on a post, a new edge is created to represent that relationship.

1.2 Graph Terminologies

Some terminologies that are frequently used in graph data structures are as follows:

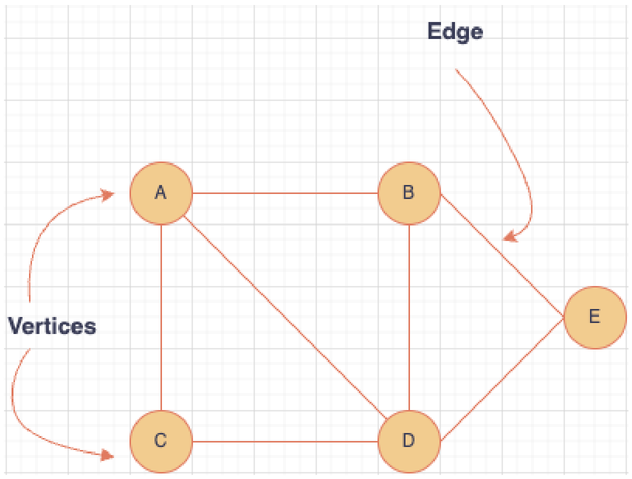


Figure 1: Simple Undirected Graph

1. **Vertex:** Vertex is also known as node. We refer to each individual data element as a node or a vertex. The vertices in the figure above are A, B, C, D, and E.

2. **Edge:** The relationship or connection between two vertices in a graph is represented by an edge. The edges in the above figure are: AB, AC, AD, BD, CD, BE, DE.

   There are 2 types of edges:

   (a) **Undirected Edge:** An undirected edge is a bidirectional edge. Vertices A and B have an undirected edge if it exists, and edge (A, B) equals edge (B, A).

   (b) **Directed Edge:** An edge that is directed is also unidirectional. Edge (A, B) is not equivalent to edge (B, A) if there is a directed edge connecting vertices A and B.

3. **Adjacency:** The relationship between vertices that are directly connected by an edge is referred to as adjacency. Vertex A and vertex B, for instance, are regarded as neighboring if they are joined by an edge.

4. **Path:** A series of vertices joined by edges in a graph is called a path. From one vertex to another, it depicts a path or route. Both simple and cyclic paths (with no repeating vertices) are possible.

5. **Weighted Edge:** An edge in a graph that has a weight connected to it is said to be weighted. The weight between the vertices connected by the edge signifies a particular attribute or cost associated with the connection.

6. **Degree:** The total number of edges connected to a vertex in a graph is called as a degree.

   There are 2 types of edges:

   (a) **Indegree:** The total number of incoming edges connected to a vertex is called as indegree.

   (b) **Outdegree:** The total number of outgoing edges connected to a vertex is called as outdegree.

7. **Self-loop:** An edge that starts and ends at the same vertex is called as a self-loop.

1.3 Types of Graphs

Below are the most common graph types seen in data structures:

1. **Undirected Graph:** A graph where each edge is bi-directional is called Undirected Graph. There is no discernible direction among the edges.

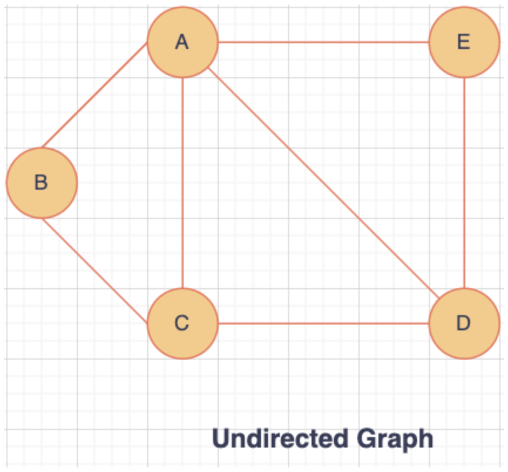


Figure 2: Undirected Graph

2. **Directed Graph:** A directed graph is also known as a digraph. In this, all the edges points in a single direction.

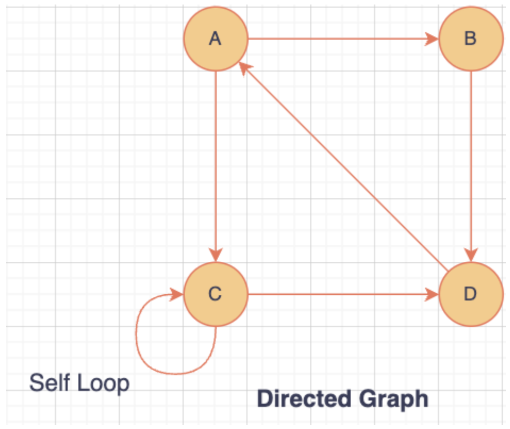


Figure 3: Directed Graph

3. **Weighted Graph:** A graph where each edge is assigned a value is called a Weighted Graph. Weights are the values that correlate to the edges. Depending on the graph, a value in a weighted graph can indicate quantities like cost, distance, and time. Weighted graphs are commonly used in computer network modeling.

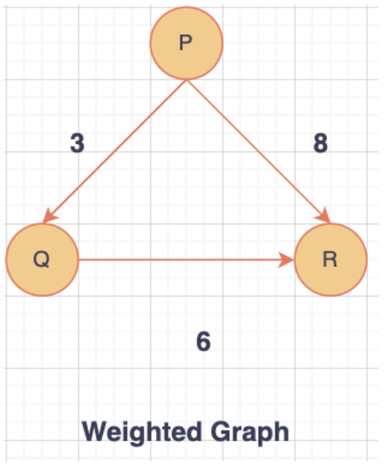


Figure 4: Weighted Graph

4. **Unweighted Graph:** A graph where the edge has no assigned weight or value is called an Unweighted Graph. By default, all of the graphs are unweighted until a value is assigned to them.

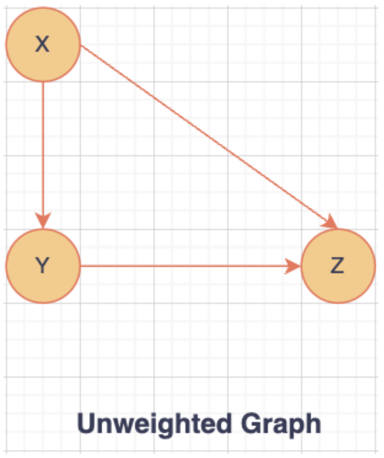


Figure 5: Unweighted Graph

5. **Cyclic Graph:** A path that starts and ends at the same node without going via any other nodes twice is called a cycle, and a cyclic graph has at least one cycle. They can be either directed or undirected.

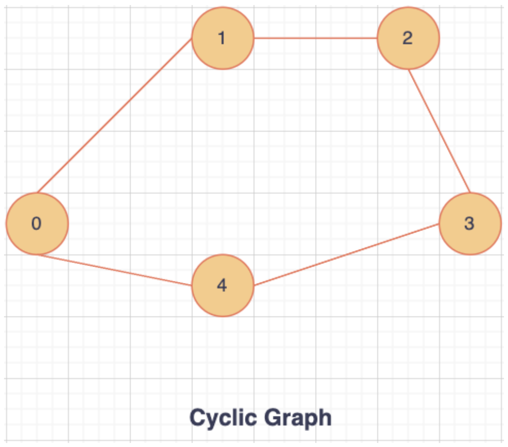


Figure 6: Cyclic Graph

6. **Acyclic Graph:** An acyclic graph does not contain any cycles. When it consists of only one connected component and no cycles, it is also referred to as a tree.

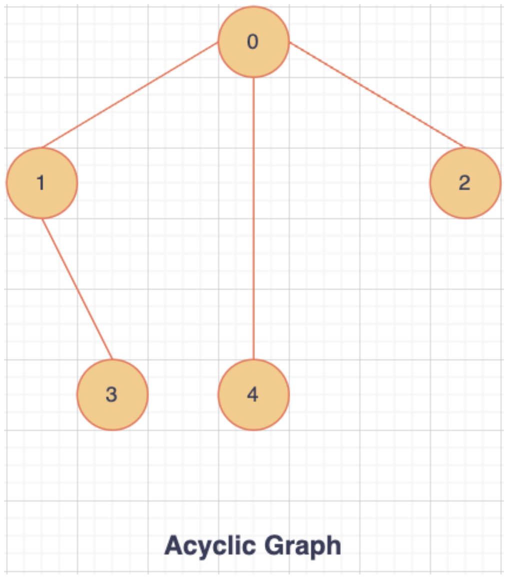
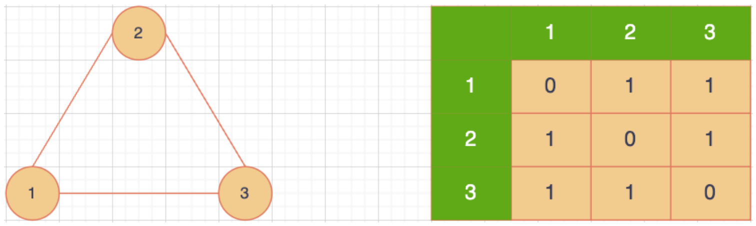


Figure 7: Acyclic Graph

1.4 Graph Representations

The two most popular methods for expressing graphs in data structures are listed below:

1. **Adjacency Matrix:**
   In this format, a graph with four vertices can be represented by a matrix of size $4X4$; that is, a graph of size total number of vertices by total number of vertices can be used to describe the graph.
   Vertices in this matrix are represented by rows and columns.
   Either 1 or 0 are present in this matrix. In this case, a 1 indicates that an edge exists between the row and column vertices, whereas a 0 indicates that none exists.
   An undirected graph's adjacency matrix is symmetric, but a digraph's adjacency matrix doesn't have to be.



	1	2	3
1	0	1	1
2	1	0	1
3	1	1	0

Figure 8: Adjacency Matrix Example

Key Points:

(a) It's recommended to use Adjacency Matrix when we have dense graphs, because it's easy to represent large-scale edges in smaller space.

(b) Since this form uses a $V * V$ matrix, $O(|V|^2)$ space is needed in the worst scenario.

(c) In order to add/remove a vertex in the $V * V$ matrix, the storage needs to be increased/reduced to $(|V|+1)^2$. We must duplicate the entire matrix in order to accomplish this. $\mathcal{O}(|V|^2)$ is the complexity as a result.

(d) In order to add/remove an edge, say, from i to j, where $matrix[i][j] = 1$, it takes $\mathcal{O}(1)$ time.

(e) The matrix's content must be examined in order to locate any edges that currently exist. It is possible to check $matrix[i][j]$ in $\mathcal{O}(1)$ time given two vertices, let's say i and j.

2. **Adjacency List:**
In this representation, a graph is represented by an array (A) of linked lists. The edges are kept as a list, and the vertices are kept as an index of the one-dimensional array. This indicates that every component in the array Ai is a list. It includes every vertex that is adjacent to vertex i.

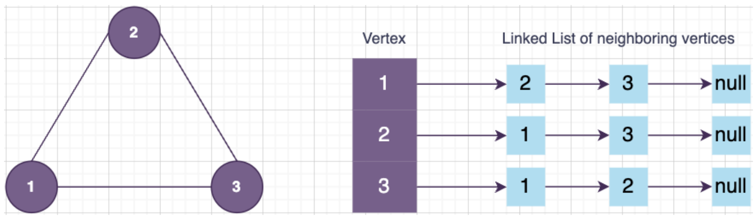


Figure 9: Adjacency List Example

Key Points:

(a) An Adjacency List is a hybrid between an Edge List and an Adjacency Matrix.

(b) It's recommended to use Adjacency List when we have sparse graphs, that is when we have number of edges as small as the number of vertices.

(c) We save the neighbors of each vertex in this representation. In the worst scenario, $\mathcal{O}(V)$ is needed for a vertex on a linked graph, and $\mathcal{O}(E)$ is needed to store the neighbors of each vertex. $\mathcal{O}(|V| + |E|)$ is the total space complexity as a result.

(d) The adjacency list has two pointers: one links to the front node and the other to the back node. As a result, a vertex/edge can be directly inserted in $\mathcal{O}(1)$ time.

(e) To remove a vertex, we must first search for it, which in the worst case will take $\mathcal{O}(|V|)$ time. Next, we must traverse the edges, which in the worst case will take $\mathcal{O}(|E|)$ time. $\mathcal{O}(|V| + |E|)$ is the total time complexity as a result.

(f) In order to remove an edge, we must go through all of the edges, in the worst situation scenario. $\mathcal{O}(|E|)$ is the time complexity as a result.

(g) Each vertex in an adjacency list has a list of neighboring vertices connected to it. To find an edge in a given graph, we must first look for vertices that are adjacent to the provided vertex. At worst, we would have to search for every neighboring vertices. A vertex can have at most $\mathcal{O}(|V|)$ neighbors. $\mathcal{O}(|V|)$ is the time complexity as a result.

1.5 Basic Graph Operations

1. Insertion or Deletion of Vertices (Nodes) in the graph.
2. Insertion or Deletion of Edges in the graph.
3. Check if the graph contains a given value.
4. Find the path from one vertex to another vertex.

1.6 Applications of Graphs

Some of the real-world applications of graphs across various fields are given below:

1. **Social network graphs:** Graph data structures are commonly employed to model social networks, including friend connections on platforms like social media.
2. **Transportation networks:** In transportation networks, such as road systems or airports, graphs are often employed to depict the connections between different locations. These graphs help visualize how various places are linked together.
3. **Neural Networks:** In neural networks, vertices symbolize neurons, while edges represent the synapses connecting them. These networks help us comprehend brain function and how connections evolve during learning.
4. **Compilers:** Graph data structures play a crucial role in compilers. They find applications in type inference, data flow analysis, register allocation, and various other tasks. Additionally, specialized compilers leverage graphs for purposes like query optimization in database languages.
5. **Robot planning:** In the context of planning paths for autonomous vehicles, graph-based representations depict the possible states the robot can occupy as vertices, with edges representing the feasible transitions between these states.

1.7 Exercise Problem

Question: Given below a Directed Graph, consisting of 5 vertices, use an Adjacency Matrix and Adjacency List to represent the graph. Also find the indegree and outdegree for each of these vertices.

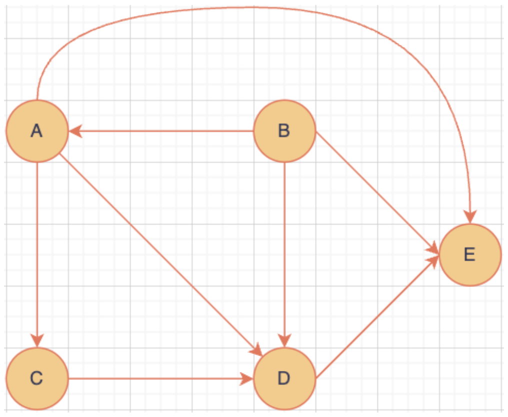


Figure 10: Directed Graph

Solution:

Given below is the Adjacency List Representation for the given Directed Graph.

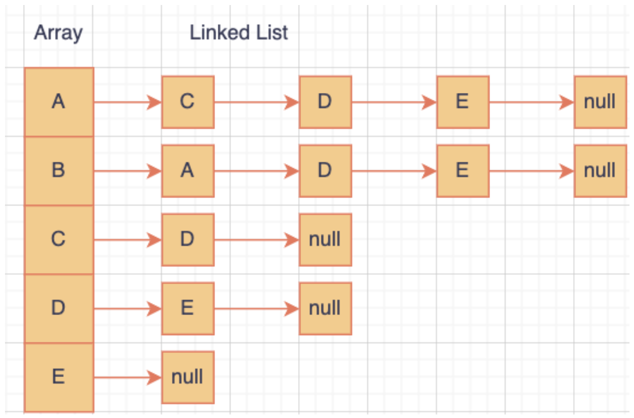
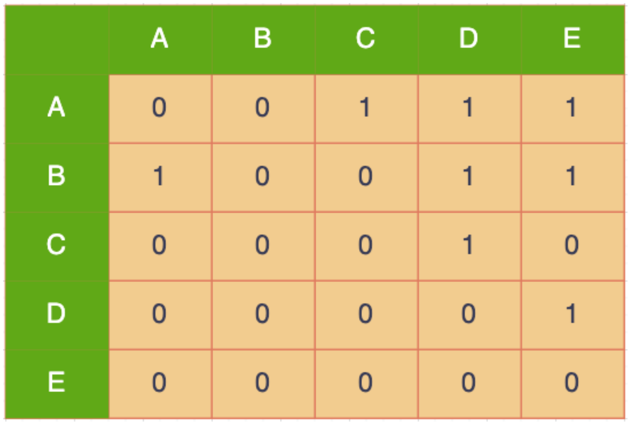


Figure 11: Adjacency List

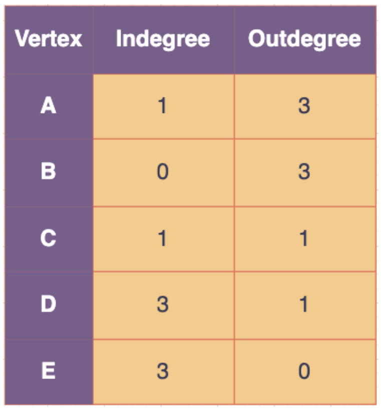
Given below is the Adjacency Matrix Representation for the given Directed Graph.



	A	B	C	D	E
A	0	0	1	1	1
B	1	0	0	1	1
C	0	0	0	1	0
D	0	0	0	0	1
E	0	0	0	0	0

Figure 12: Adjacency Matrix

Given below is the Indegree/Outdegree for each of the vertices of the given Directed Graph.



Vertex	Indegree	Outdegree
A	1	3
B	0	3
C	1	1
D	3	1
E	3	0

Figure 13: Indegree-Outdegree of Vertices

Chapter 2

Depth-first search, search trees, edge types, pre/post

2.1 Introduction

Before diving deep into our topic, let's first understand the basics of it.

A graph is a non-linear data structure composed of vertices (also known as nodes) connected by edges. Vertices represent entities, and edges represent relationships between those entities. A graph may have cycles.

A tree is a hierarchical structure where nodes are connected by edges. Each node can have multiple children but only one parent. The topmost node is known as the root. A tree doesn't contain any cycle.

Graphs can be disconnected (i.e., have multiple components), while trees are always connected.

Now, let's understand some basic terminologies related to tree data structure.

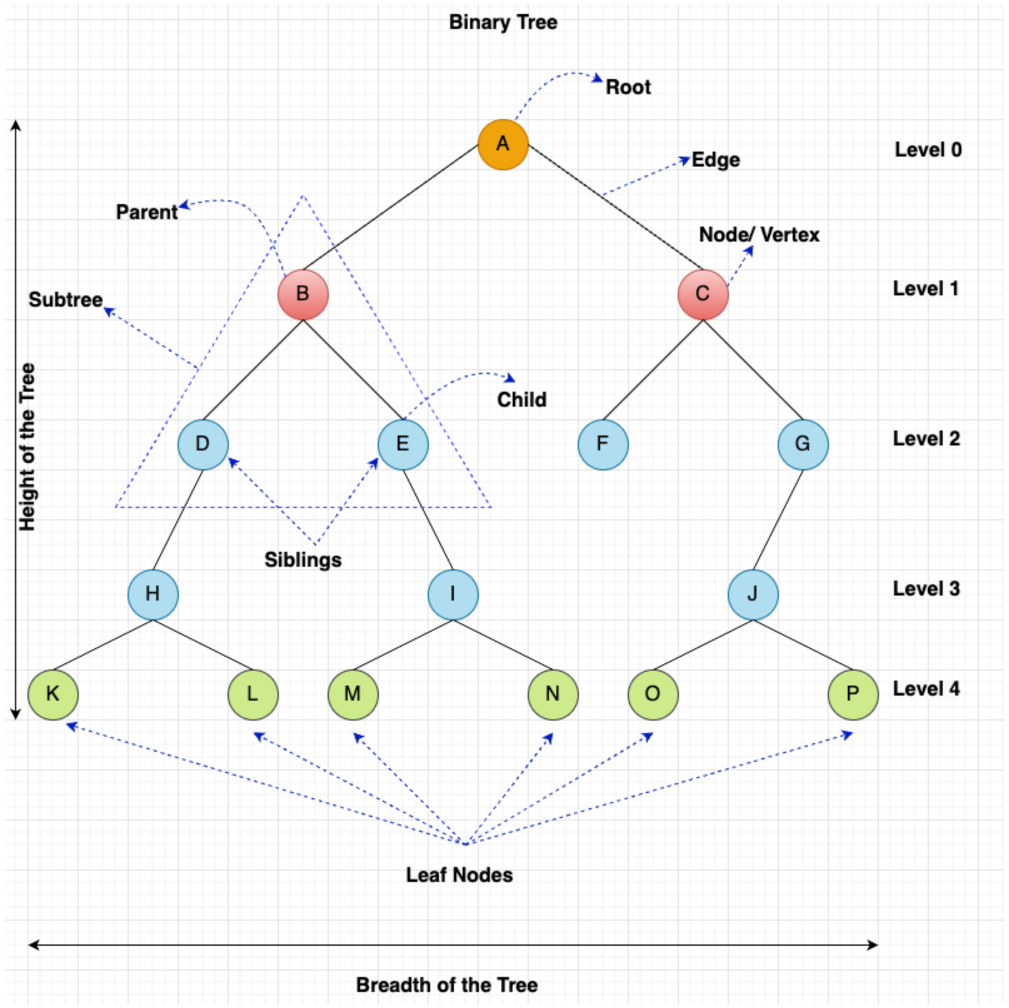


Figure 14: Basic Graph Terminologies.

1. **Node (Vertex):** A node, also known as a vertex, represents an individual element in a graph. It can hold a value, condition, or simply serve as a point of connection.
2. **Root Node:** The root node is the topmost node in a tree-like structure. It acts as the prime ancestor from which all other nodes descend. In a tree, there is only one root node.
3. **Child Node:** A child node is directly connected to another node when moving away from the root. It is an immediate descendant of its parent. In a tree, each node (except the root) has exactly one parent but can have multiple children.
4. **Parent Node:** A parent node is an immediate ancestor of a given node. Again, in a tree, each node (except the leaves) has exactly one parent.
5. **Leaf Node:** A leaf node is a node with no children. It's like the end of a branch in a tree. In other words, a leaf node does not have any outgoing edges.
6. **Edge:** An edge represents the connection between two nodes (vertices).
7. **Depth:** The depth of a node is the distance between that node and the root (topmost node) of the tree or graph.
8. **Level:** The level of a node is determined by the number of edges between that node and the root, plus one. The root node is at level 1, its immediate children are at level 2, and so on.

9. **Height:** The height of a node is the number of edges on the longest path from that node to a descendant leaf (a node with no children).
10. **Breadth:** The breadth refers to the number of leaves (nodes with no children) in a tree or subtree. It reflects the width or spread of the tree.
11. **Subtree:** A subtree is a tree that consists of a specific node (called the root of the subtree) and all its descendants. In other words, it's a smaller tree within a larger one.
12. **Binary Tree:** A binary tree is a tree data structure where each node has at most two children: a left child and a right child.

Tree Traversal

So what does it mean to traverse a tree?

Trees are a type of graph and tree traversal is also known as graph traversal/tree search. However, the process of traversing through a tree is a little different than the more broad process of traversing through a graph.

When traversing a tree, our primary goals are typically to either check all the nodes in the tree structure or update them. It's important to note that we don't search through the nodes of a tree more than once. If we're trying to check or update every single node in a tree, we avoid revisiting nodes to prevent redundancy.

It's not just about visiting each node only once, but the order in which we traverse matters a lot!

When dealing with trees, there are primarily two techniques for traversing and visiting each node exactly once. Our choices boil down to going wide (breadth-first traversal) or going deep (depth-first traversal).

Thus, we have 2 methods for tree traversal $\longrightarrow$

1. Breadth-First Search
2. Depth-First Search

In this chapter, we would be learning Depth-First Search in detail.

Depth-First Search

In Depth-First Search (DFS), we follow a path until we reach the end. Specifically, we traverse through one branch of a tree until we reach a leaf node, and then we backtrack to the parent node to follow another path (branch) to the remaining leaf nodes.

Depth-First Search Strategies

We have 3 different DFS traversal techniques.
Consider the below graph to demonstrate Preorder/ Inorder/ Postorder Traversal.

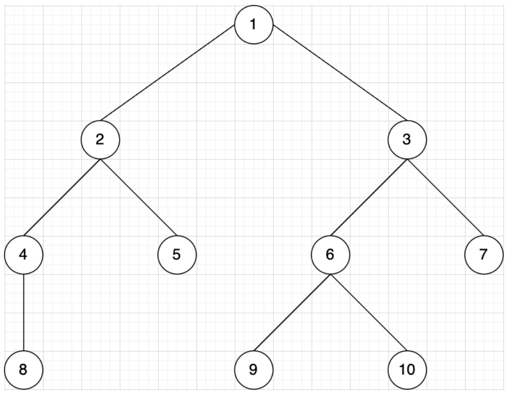


Figure 15: Simple Graph for Traversal Methods.

1. **Preorder Traversal:** In preorder traversal, we visit the root node first, then recursively traverse the left subtree, and finally the right subtree.
   $[root] \rightarrow [left] \rightarrow [right]$
   Preorder traversal is commonly used to serialize or make a replica of the tree.

   From the above figure, the Preorder Traversal is as follows:
   $1 \rightarrow 2 \rightarrow 4 \rightarrow 8 \rightarrow 5 \rightarrow 3 \rightarrow 6 \rightarrow 9 \rightarrow 10 \rightarrow 7$

2. **Inorder Traversal:** In inorder traversal, we visit the left subtree first, then the root node, and finally the right subtree.
   $[left] \rightarrow [root] \rightarrow [right]$
   Inorder traversal in binary search trees (BSTs) returns nodes sorted (in ascending order).

   From the above figure, the Inorder Traversal is as follows:
   $8 \rightarrow 4 \rightarrow 2 \rightarrow 5 \rightarrow 1 \rightarrow 9 \rightarrow 6 \rightarrow 10 \rightarrow 3 \rightarrow 7$

3. **Postorder Traversal:** In postorder traversal, we visit the left subtree first, then the right subtree, and finally the root node.
   $[left] \rightarrow [right] \rightarrow [root]$
   Postorder traversal is frequently used in expression trees to evaluate expressions and delete nodes.

   From the above figure, the Postorder Traversal is as follows:
   $8 \rightarrow 4 \rightarrow 5 \rightarrow 2 \rightarrow 9 \rightarrow 10 \rightarrow 6 \rightarrow 7 \rightarrow 3 \rightarrow 1$

2.2 Applications of DFS

1. **Solving Puzzles with One Solution (e.g., Mazes):** DFS can be adapted to find a single solution to a maze. By only including nodes on the current path in the visited set, you can explore the maze until you reach the exit. This approach ensures that you find a valid path without revisiting nodes.

2. **Web Crawlers:** Web crawlers use DFS to explore and index web pages.

3. **Maze Generation:** DFS can be used to generate random mazes.

4. **Model Checking:** DFS can explore the state space of the model, checking for desired conditions or detecting errors.

2.3 Pseudocode Algorithm of DFS

```
procedure dfs(G):

for all $v \in V$:
visited(v) = false

for all $v \in V$:
if not visited(v) = explore(v)
```

Given below are the steps to follow to traverse a graph using DFS:

1. Start at a source node: Select a starting node (which is typically the root in a tree or an arbitrary node in a graph).
2. Explore as deep as possible: From the current node, visit the neighboring unvisited nodes that are nearby. Apply DFS to these neighbors recursively.
3. Backtrack when necessary: Go back to the previous node and investigate the other branches if there are no unexplored neighbors.
4. Repeat until all nodes are visited: Continue in this manner until you have visited every node or until you have achieved your goal.

2.4 Types of Edges in DFS

In graph theory, we come across many different types of edges when talking about depth-first search (DFS). Now let's explore them:

1. **Tree Edges:** The edges that make up the DFS tree itself are called tree edges. A tree edge is created when we traverse from a parent node to its unvisited child node. These edges link nodes in the order that they are visited and define the DFS tree's structure.
2. **Back Edges:** In the DFS tree, back edges link a node to one of its ancestors. They show that there is a cycle present in the graph. The graph is neither a forest nor a tree if there is a back edge. In DFS, back edges are essential for cycle detection.
3. **Forward Edges:** In the DFS tree, forward edges link a node to one of its descendants. They appear when we move from a parent node to a child node and then use a different path to return to the same child node afterwards.
4. **Cross Edges:** In the DFS tree, cross edges link nodes that are neither descendants nor ancestors. They appear when we explore the graph's unconnected branches. As a result, they lead to a node that has been fully explored (already postvisited).

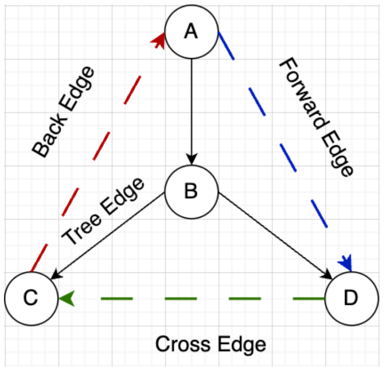


Figure 16: DFS Tree showing Edge Types.

2.5 Previsit and Postvisit Orderings in DFS

When using Depth First Search (DFS), every vertex in a graph is marked as visited. Therefore, some additional details can also be stored to make DFS useful. For example, the DFS traversal order at which the vertices are visited.

The additional information that may be stored while doing a DFS on a graph and that proves to be quite helpful are the Pre-visit and Post-visit numbers.

Now let's delve into the concepts of previsit and postvisit orderings in depth-first search (DFS):

1. **Previsit Order:** Previsit order (discovery time), is assigned to a node when we visit it for the first time during DFS. The nodes that are encountered are represented by this order.
   Previsit order helps determine the order in which nodes are explored and helps to understand the traversal process.

2. **Postvisit Order:** Postvisit order (finish time), is assigned to a node when we have finished exploring that node (i.e., after visiting all its neighbors and backtracking).
   The order of postvisit signifies the completion of nodes.

Let's see the below example to understand the pre/post values in a DFS graph.

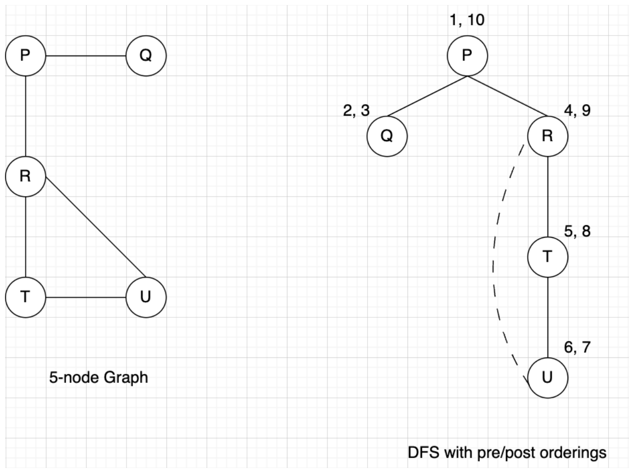


Figure 17: DFS with Pre/Post Order.

Note that, the pre/post values of the children nodes are always in between the range of the parent node.

2.6 Source and Sink

In a Directed Acyclic Graph (DAG), we can define source and sink nodes as follows:

1. **Source:** A source is a node that has no edges originating from it. In other words, it is a node without any incoming edges.
   The source is usually the traversal's starting point when executing a depth-first search (DFS) on a graph.

2. **Sink:** A node to which no edges lead is called a sink. This node has no outgoing edges.
   The traversal endpoint is known as the sink in DFS. You don't explore further down from a sink after you've reached it.

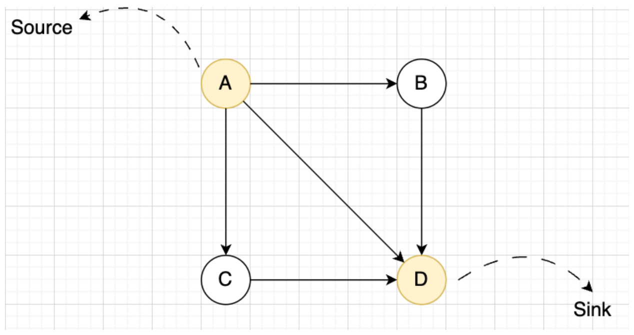


Figure 18: Directed Acyclic Graph with source and sink.

Consider a directed acyclic graph as shown above. In this example, we can indicate A as the source node, because it has no incoming edges, while 'D' is indicated as the sink node, because it has only incomimg edges but no outgoing edge.

2.7 Time and Space Complexity of DFS

1. The time complexity of DFS for all the 3 cases (best-case, average-case, worst-case) is same i.e., $\mathcal{O}(V + E)$, where V is the number of vertices and E is the number of edges.
2. Since the Depth-First Search (DFS) algorithm uses a recursion stack or visited array, its auxiliary space complexity is $\mathcal{O}(V)$, where V is the number of vertices in the graph.

2.8 Exercise Problem

Question: Perform depth-first search on the following graph; whenever there's a choice of vertices to explore, pick the one that is alphabetically first. Also, do the following tasks:

1. Classify the edge types as forward edges, back edges, and cross edges.
2. Store the pre and post number of each vertex.
3. What are the sources and sinks of the graph?

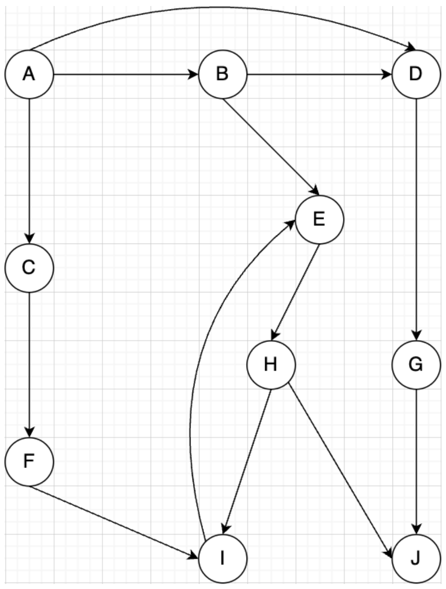


Figure 19: Simple Directed Graph

Solution:

The DFS for the above graph is given below as:

$A \to B \to D \to G \to J \to E \to H \to I \to C \to F$

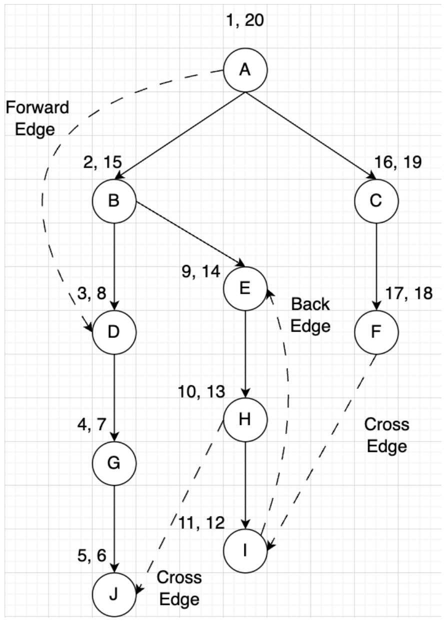


Figure 20: DFS on a Directed Graph with pre/post values.

From the above DFS Graph, we could classify one Forward Edge $A \to D$, one Backward Edge $I \to E$, and two Cross Edges $H \to J$ and $F \to I$.

Also, we have one source node i.e., A (with no incoming edges) and one sink node i.e., J (with no outgoing edges).

2.9 Coding Problem

Question: Given a reverse polish expression tree, evaluate the expression. Return an integer that represents the value of the expression.

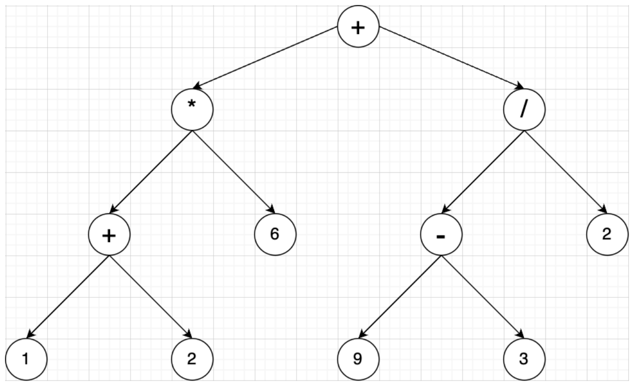


Figure 21: Expression Tree

Solution:

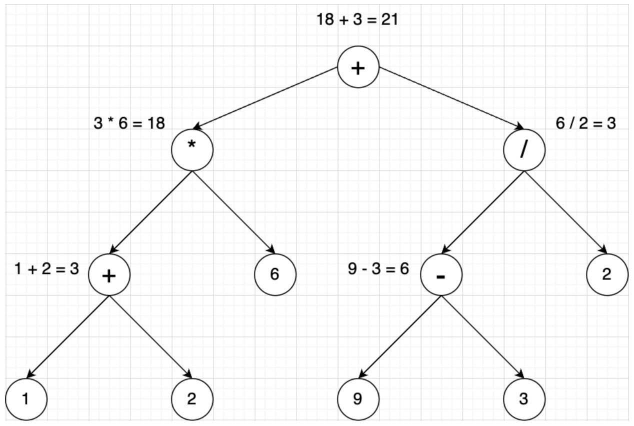


Figure 22: Evaluated Expression Tree

Post-order traversal is used to evaluate the expression.
It is of the form $left\_node \rightarrow right\_node \rightarrow root\_node$
After looking at the above "Evaluated Expression Tree", we can see that the final result is 21.

Please refer the Python file 'expression.py' for complete solution.

Chapter 3
Shortest path: Dijkstra's algorithm

3.1 Introduction

Until now, we have learnt the fundamentals of graph theory and the traversal methods like for example depth-first search. We already know that graphs can be directed, or undirected, and may even contain cycles. While understanding the graph theory, we came across Weighted Graphs.

It doesn't really matter if a graph is directed, undirected, or has cycles—a weighted graph is still fascinating. Fundamentally speaking, a weighted graph is one in which each edge is assigned a value. An edge's "weight" comes from the value associated with it. It can also be referred to as cost/distance/capacity interchangeably.

In this topic, we will be studying one of the shortest path algorithm i.e., Dijkstra's Algorithm.

We have already explored one of the traversal methods i.e., Breadth-First-Search (BFS). And Dijkstra's Algorithm employs similar approach to BFS to find the shortest path. Now, let's recall the BFS method.

While exploring the neighboring nodes, BFS starts from the source node and explores nodes layer-by-layer. The purpose of BFS is to find the shortest path in an unweighted graph, and it treats all edges equally. It doesn't use priority queue and instead uses level-order traversal technique to explore it's neighbors.

While, Dijkstra's Algorithm also starts it's exploration from the source node, it explores it's neighbors based on edge weights. It finds the shortest path in a weighted graph (non-negative edges). And it maintains the priority queue to select the next node with minimum edge weight.

Dijkstra's Algorithm

Dijkstra's algorithm, introduced by Dutch computer scientist Edsger Dijkstra in 1959, is applicable to weighted graphs. These graphs can be either directed or undirected. However, a crucial requirement for using the algorithm is that every edge weight must be nonnegative.

Dijkstra's algorithm computes the shortest path from a single source node to all other nodes in a graph, assuming that those nodes are reachable from the starting node. When finding a path from an initial vertex to a destination vertex, the goal is to minimize the overall "cost" associated with that path. The algorithm employs a greedy strategy, seeking the next optimal solution with the hope that it ultimately leads to the best overall solution for the entire problem.

After running Dijkstra's algorithm once and finding the shortest path between reachable nodes, we can save the results. Subsequently, we can look up these results without re-executing the algorithm.

Example

Let me illustrate this through a simple example to understand it better!

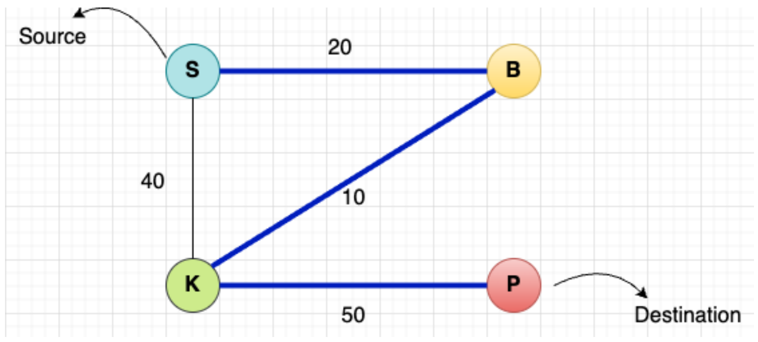


Figure 23: Simple Graph Example

Suppose, we want to travel from Seattle (source) to Port Angeles (destination). There are two ways we could reach our destination - one is by ferry, and the other is by road.

The vertices/nodes are the cities → Seattle (S), Bainbridge Island (B), Kingston (K), Port Angeles (P) .

The edges are the connections and the weights are the distances between each cities →
Seattle to Bainbridge Island (by ferry) with weight 20 miles.
Seattle to Kingston (by road) with weight 30 miles.
Bainbridge Island to Kingston (by ferry) with weight 10 miles.
Kingston to Port Angeles (by road) with weight 50 miles.

Our goal is to find the shortest path from Seattle (S) to Port Angeles (P). Let's apply Dijkstra's algorithm here -

1. **Initialize distances:**
   Distance from S to itself: 0 (starting point).
   Distance from S to B: 20 (ferry to Bainbridge Island).
   Distance from S to K: 40 (road to Kingston).
   Distance from S to P: Infinity (unknown).

2. **Select the vertex with the minimum distance:**
   We will choose Bainbridge Island (B), since it's the closest (distance = 20).

3. **Update distances:**
   Distance from S to K via B: $20 + 10 = 30$ (ferry to Kingston, then road to Port Angeles).

4. Next, select Kingston (K) as the closest vertex (distance = 30)

5. **Update distances:**
   Distance from S to P via K: $30 + 50 = 80$ direct road to Port Angeles).

6. Finally, we have:
   Distance from S to P: 80 (via Bainbridge Island) $<$ 90 (via Kingston).

So, the shortest path from Seattle to Port Angeles is via Bainbridge Island, covering a total distance of 80 miles (ferry + road).

3.2 Applications of Dijkstra's Algorithm

Dijkstra's Algorithm has several real-world use cases, some of which are as follows:

1. **Geographical Maps:** Dijkstra's algorithm determines the least distance along the path while determining the shortest route between two points on a map. It serves as the foundation for navigation systems' route planning.

2. **Social Networking Applications:** The shortest path between users can be found quickly and effectively using Dijkstra's algorithm based on connections.
3. **Telephone Networks:** Every line in a telephone network has a bandwidth. Dijkstra's algorithm assists in determining the shortest path between switching stations by representing cities as vertices and transmission lines as edges.
4. **Robotic Path:** This algorithm module is loaded into automated drones and robots that are used to deliver packages to designated locations or for specific tasks. Once the source and destination are known, the robot or drone moves in an ordered manner by taking the shortest path possible to continue delivering the package in the shortest amount of time.

3.3 Pseudocode Algorithm

```
procedure dijkstra (G, s):

Input: Graph $G = (V, E)$, directed or undirected; positive edge weights; vertex $s \in V$

Output: For all vertices u reachable from s, dist[u] is set to the distance from s to u.

for all $u \in V$
$dist[u] = \infty$
$prev[u] = nil$
$dist[s] = 0$

Q = $\{s\}$
while Q is not empty
u = vertex in Q with smallest dist[u]
remove u from Q

for each neighbor v of u
if (dist[u] + weight(u, v)) < dist[v]
dist[v] = dist[u] + weight(u, v)
prev[v] = u
add v to Q
```

Let's break down the pseudocode for Dijkstra's algorithm step-by-step:

1. Initialize two arrays.
   dist[u]: Stores the shortest distance from the source vertex *s* to vertex *u*.
   Set all distances to infinity except $dist[s] = 0$.
   prev[u]: Keeps track of the previous vertex on the shortest path to u. Initialize all to nil.
2. Create an priority queue Q with Source vertex. This queue will track vertices to process in order of minimum cost/distance.
3. While the queue is not empty, select the vertex *u* from Q with the smallest tentative distance (dist[u]).
   Also, remove *u* from Q.
4. For each neighbor v of u:
   Calculate the tentative distance from *s* to *v* through *u*: dist[v] = dist[u] + weight(u, v).
   If this distance is smaller than the current dist[v], update dist[v] and set prev[v] = u.
5. Repeat steps 2, 3, and 4 until Q is empty or until the destination vertex is reached.
6. The dist[] array now contains the shortest distances from the source vertex s to all other vertices.
   The prev[] array stores the previous vertex on the shortest path from *s* to each vertex.

3.4 Time and Space Complexity of Dijkstra's Algorithm

1. In the best-case scenario, the algorithm efficiently finds the shortest paths, with the priority queue operations optimized, leading to the overall time complexity of $\mathcal{O}((V+E)\log V)$.
   This kind of situation usually occurs in sparse graphs, which have a comparatively small number of edges relative to vertices.
2. The average-case time complexity is usually $\mathcal{O}((V+E)\log V)$, which is also the best-case scenario's time complexity.
   The reason for this is because most real-world graphs, which are frequently neither highly sparse nor fully connected, respond favorably to Dijkstra's method.
3. In the worst-case scenario, the time complexity is $\mathcal{O}((V^2)\log V)$.
   This happens when there is a lack of optimization and the graph is dense, with plenty of edges, making the priority queue operations less effective.
4. Dijkstra's algorithm's auxiliary space complexity varies from $\mathcal{O}(V)$ to $\mathcal{O}(E+V)$ based on the data structures and implementation.
   Given that $E \geq V-1$ in the majority of instances, we usually regard it as $\mathcal{O}(E)$.

3.5 Exercise Problem

Question: For the following graph, starting at node A, run Dijkstras's Algorithm. Also, perform the given tasks:

1. Draw a table to show the intermediate distance values at each iteration of the algorithm.
2. Show the final shortest-path.

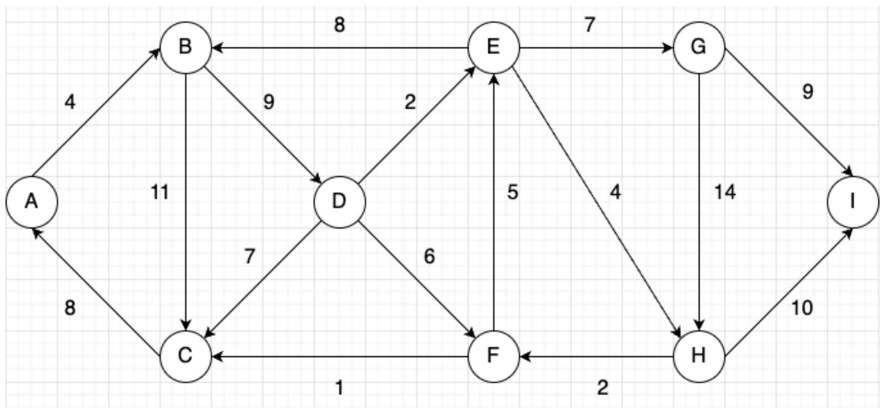


Figure 24: Directed Graph for computing Shortest Path

Solution:

Let's apply Dijkstra's Algorithm to the given weighted graph. We'll start at node A and follow these steps:

1. **Initialization:**
   $\rightarrow$ Set the initial distance to zero for node A and infinity for all other nodes.
   $\rightarrow$ Mark all nodes as unvisited and create an unvisited set containing all nodes.
   $\rightarrow$ Assign distances: 0 for A, infinity for others.
2. **Iteration1:**
   $\rightarrow$ Consider neighbors of A: B(4).
   $\rightarrow$ B is the only tentative distance from A: $B = 4$.

3. **Iteration2:**
   $\rightarrow$ Select B (smallest tentative distance).
   $\rightarrow$ Consider neighbors of B: C(11), D(9).
   $\rightarrow$ Update tentative distance from B(4): C $(4 + 11 = 15)$, and D $(4 + 9 = 13)$.

4. **Iteration3:**
   $\rightarrow$ Select D (smallest tentative distance).
   $\rightarrow$ Consider neighbors of D(13): C(7), E(2), F(6).
   $\rightarrow$ Update tentative distance from D(13): C $(13 + 7 = 20)$, as $20 > 15$, we are not updating C. So, C remains as it is $(C = 15)$.
   Now, E $(13 + 2 = 15)$, and F $(13 + 6 = 19)$.

5. **Iteration4:**
   $\rightarrow$ Select C (smallest tentative distance).
   $\rightarrow$ Since node C can directly reach node A, and A is the source node (minimum distance of 0), there are no updates needed for C.

6. **Iteration5:**
   $\rightarrow$ Select E (next smallest tentative distance).
   $\rightarrow$ Consider neighbors of E(15): B(8), G(7), H(4).
   $\rightarrow$ Update tentative distance from E(15): B $(15 + 8 = 23 > 4)$, as current_ distance > previous_distance. Hence, no change for B).
   Now, G $(15 + 7 = 22)$, and H $(15 + 4 = 19)$.

7. **Iteration6:**
   $\rightarrow$ Select F (smallest tentative distance).
   $\rightarrow$ Consider neighbors of F(19): C(1), E(5).
   $\rightarrow$ Update tentative distance from F(19): No updates are required as C and E already have the minimum distances; C $(19 + 1 = 20 > 15)$, and E $19 + 5 = 24 > 15$.

8. **Iteration7:**
   $\rightarrow$ Select H (smallest tentative distance).
   $\rightarrow$ Consider neighbors of H(19): F(2), I(10).
   $\rightarrow$ Update tentative distance from H(19): From H to C $(19 + 2 = 21 > 19)$, hence no update required here.
   From H to I $(19 + 10 = 29)$, gets updated.

9. **Iteration8:**
   $\rightarrow$ Select G (smallest tentative distance).
   $\rightarrow$ Consider neighbors of G(22): H(14), I(9).
   $\rightarrow$ Update tentative distance from G(22): From G to H $(22 + 14 = 36 > 19)$; From G to I $(22 + 9 = 31 > 29)$.
   Hence no update required here.

10. **Iteration9:**
    $\rightarrow$ Select I (only remaining unvisited node).
    $\rightarrow$ No neighbors to consider.

Now let's create the intermediate distance table:

N	A	B	C	D	E	F	G	H	I
1	0	inf	inf	inf	inf	inf	inf	inf	inf
2	0	4	15	13	inf	inf	inf	inf	inf
3	0	4	15	13	15	19	inf	inf	inf
4	0	4	15	13	15	19	inf	inf	inf
5	0	4	15	13	15	19	22	19	inf
6	0	4	15	13	15	19	22	19	inf
7	0	4	15	13	15	19	22	19	29
8	0	4	15	13	15	19	22	19	29
9	0	4	15	13	15	19	22	19	29

Figure 25: Table showing Intermediate Distances

Here is the shortest path from A to I as follows:
$A \to B \to D \to E \to H \to I$ (Total distance: $4 + 9 + 2 + 4 + 10 = 29$)

$$\begin{tikzcd}
 & B \arrow[dr, "9"] & & E \arrow[ddr, "4"] & & \\
A \arrow[ur, "4"] & & D \arrow[ur, "2"] & & & I \\
 & & & & H \arrow[ur, "10"] &
\end{tikzcd}$$

Figure 26: Final Shortest Path

3.6 Coding Problem

Question: There are n cities connected by some number of flights. You are given an array flights where flights[i] = $[from_i, to_i, price_i]$ indicates that there is a flight from city $from_i$ to city to_i with cost $price_i$.
You are also given three integers src, dest, and k, return the cheapest price from src to dest with at most k stops. If there is no such route, return -1.

Solution:

Below is the graph showing the cities connected by number of flights.

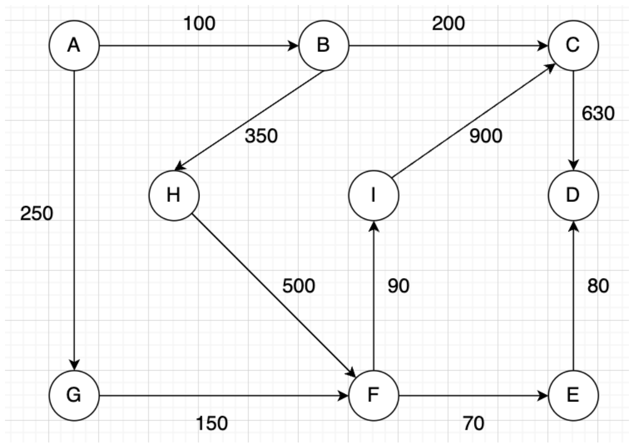


Figure 27: Graph showing cities and its flight routes.

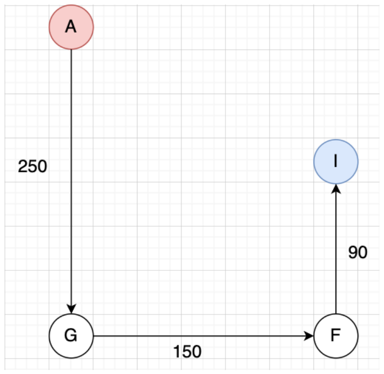


Figure 28: Cheapest Flight from city A to city I with max_stop = 5.

Given: Input $\rightarrow$ max_stop$(k = 5)$,
Flights $\rightarrow \{[A, G, 250], [G, F, 150], [F, I, 90]\}$ with 2 stops.
Total Minimum Distance from City A to City I with 5 max stops = 490.

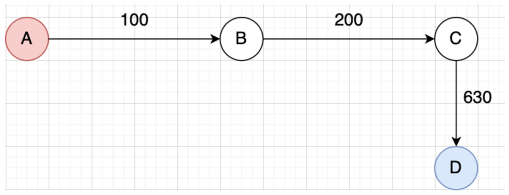


Figure 29: Cheapest Flight from city A to city D with max_stop = 2.

Given: Input $\rightarrow$ max_stop$(k = 2)$,
Flights $\rightarrow \{[A, B, 100], [A, C, 200], [C, D, 630]\}$ with 2 stops.

Total Minimum Distance from City A to City D with 2 max stops = 930.

We could achieve this with minimum distance of 550 with 3 stops. But, as we have to reach D in 2 stops, we get the alternative minimum distance of 930.

Since, we use minimum priority queue, we first get the minimum distance of 550 with 3 stops, But stops are more than the required. So, we continue the process and remove the next entry from the queue until we get the dest node D. This entry has the distance of 930 with 3 stops.

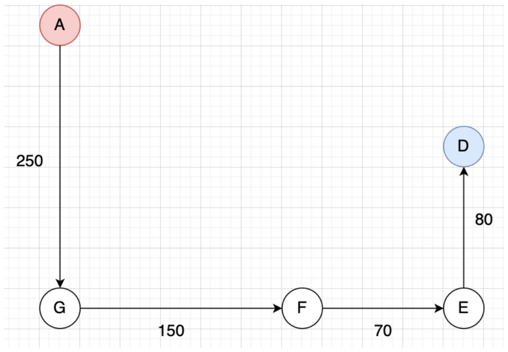


Figure 30: Cheapest Flight from city A to city D with max_stop = 3.

Given: Input $\rightarrow$ max_stop$(k = 3)$,
Flights $\rightarrow \{[A, G, 250], [G, F, 150], [F, E, 70], [E, D, 80]\}$ with 3 stops.

Total Minimum Distance from City A to City D with 3 max stops = 550.

Please refer the Dijkstras_Algorithm.py for complete solution.

Chapter 4

Longest Increasing Subsequence

4.1 Introduction

In this chapter we will see how to find the longest increasing subsequence. But before that, let's see the approach behind it.

We use Dynamic Programming when solving such problems. Dynamic programming is an algorithmic approach that tackles complex problems by dividing them into smaller, overlapping subproblems. It then stores the solutions to these subproblems for efficient computation. This technique is especially useful when problems exhibit optimal substructure.

What is Longest Increasing Subsequence (LIS)?

Given an unsorted array, we want to find the length of the longest subsequence where the elements are in ascending order (from lowest to highest). Note that the subsequence doesn't have to consist of consecutive elements from the original array, and the solution for the Longest Increasing Subsequence (LIS) problem may not always be unique.

Let us try to understand this through an example.

Consider the elements $\{-4, 10, 5, 12, 20\}$. We can identify the following increasing subsequences:
$\{-4, 10, 12, 20\}$
$\{-4, 5, 12, 20\}$
$\{10, 12, 20\}$
$\{5, 12, 20\}$
$\{12, 20\}$
The longest increasing subsequence is $\{-4, 5, 12, 20\}$ with a length of 4. However, $\{-4, 10, 12, 20\}$ is also a valid solution with the same length.

4.2 Applications of Longest Increasing Subsequence

Given below are the practical applications of LIS:

1. **Patience Diff Algorithm:** Patience Diff identifies lines that are unique to both documents, arranges them based on their appearance in the second document, and then computes the Longest Increasing Subsequence (LIS). This approach leads to a more accurate difference (diff) output.

2. **Version Control Systems:** LIS assists in version control systems by identifying the longest increasing subsequence of lines between different file versions. This process helps determine which lines were added, modified, or deleted during code changes.

4.3 Pseudocode Algorithm

```
DPLIS(array):
Input→ array: The input array of size n.
Output→ LIS: The length of the longest increasing subsequence.

n ← size of array
Lis ← array of 1s of size n

for i ← 1 to n:
for j ← 0 to $i - 1$:
if array[i] > array[j] and $Lis[i] \leq Lis[j]$:
Lis[i] ← Lis[j] + 1

return max(Lis[0], Lis[1], ..., Lis[n])
```

Let's break down the pseudocode for the Longest Increasing Subsequence (LIS) algorithm step by step:

1. **Input:** Given an array called "array" containing 'n' elements, we aim to find the length of the longest increasing subsequence within this array.
2. **Initialization:** Initialize an array called 'Lis' with a size of 'n', where each element is initially set to 1. The value at each index 'i' in 'Lis' represents the length of the Longest Increasing Subsequence (LIS) ending at position 'i'.
3. **Dynamic Programming Approach:** We traverse the array from left to right (from index 1 to n). For each element at index 'i', we compare it with all preceding elements at indices 'j' (where j < i). If array[i] is greater than array[j] and Lis[i] is less than or equal to Lis[j], we update Lis[i] to be Lis[j] + 1. This process extends the Longest Increasing Subsequence (LIS) ending at 'j' by including the element at 'i'.
4. **Final Result:** By processing all the elements, we determine the maximum value in the 'Lis' array. This maximum value corresponds to the length of the longest increasing subsequence in the original array.

Let's apply the algorithm to find LIS step-by-step through a simple example.

Suppose we have the array $[3, 10, 2, 1, 20]$.

Initially, Lis is $[1, 1, 1, 1, 1]$.

After processing each element:
For $i = 2$, we compare 10 with 3 (at index 1). Since $10 > 3$, we update Lis[2] to 2.
For $i = 3$, we compare 2 with 10 (at index 2). Since $2 < 10$, we skip this step.
For $i = 4$, we compare 1 with 10 (at index 2). Since $1 < 10$, we skip this step.
For $i = 5$, we compare 20 with 10 (at index 2). Since $20 > 10$, we update Lis[5] to 3.

The final Lis array is $[1, 2, 1, 1, 3]$, and the maximum value is 3, which is the length of the LIS $[3, 10, 20]$.

4.4 Time and Space Complexity of LIS

1. The dynamic programming approach for solving the Longest Increasing Subsequence (LIS) problem has a time complexity of $\mathcal{O}(n^2)$, where 'n' represents the number of elements in the input array. This complexity arises due to the nested loops involved in comparing elements and updating the dynamic programming (DP) array.
2. The space complexity of the dynamic programming approach is $\mathcal{O}(n)$. This space is primarily utilized to store the DP array, which tracks the lengths of increasing subsequences ending at each position in the array.

4.5 Exercise Problem

Question: Given an array of integers $[10, 22, 9, 33, 21, 50, 41, 60, 80]$, find the length of the longest increasing subsequence. Also, show the maximum subsequence of that length.

Solution:

The objective is to identify the longest subsequence within a given sequence where the elements are arranged in ascending order (from lowest to highest), and this subsequence should be as lengthy as possible.

Given an input array $[10, 22, 9, 33, 21, 50, 41, 60, 80]$, let's find out the longest increasing subsequence step-by-step.

1. **Step1:** The base case for the given problem is $L(0) = 1$. Each element L[i] will represent the length of the LIS ending at index i. Set L[i] to 1 because a single element is always an increasing subsequence of length 1.
2. **Step2:** Iterate through the list from left to right (for each index i):
   For each index i, compare it with all indices before it (from j = 0 to $i - 1$).
   If arr[j] < arr[i], update L[i] as max(L[i], L[j] + 1). This means that we extend the LIS ending at index j by including the element at index i.
3. **Step3:**
   $L[1] = max(L[1], L[0] + 1) = 2$(since $22 > 10$)

   $L[2] = 1$ (since $9 < 10$ and $9 < 22$. Skip index 0 and 1)

   $L[3] = max(L[3], L[1] + 1) = 3$ (since $33 > 22$)
   index $0 : 10 < 33$, update $L[3] = max(L[3], L[0] + 1) = 2$
   index $1 : 22 < 33$, update $L[3] = max(2, L[1] + 1) = 3$
   index 2: $9 < 33$, update $L[3] = max(3, L[2] + 1) = 3$ (here $L[2] = 1$, therefore $max(3, 1 + 1) = 3$. So, L[3] did not update as it already had max value.

   $L[4] = max(L[4], L[2] + 1) = 2$ (since $21 < 22$ and $21 < 33$, skip index 1 and 3). Possible subsequence $[10, 21]$ and $[9, 21]$

   $L[5] = max(L[5], L[3] + 1) = 4$ (since $50 > 33$)

   $L[6] = max(L[6], L[3] + 1) = 4$ (since $33 < 41$)

   $L[7] = max(L[7], L[5] + 1) = 5$ (since $60 > 50$)

   $L[8] = max(L[8], L[7] + 1) = 6$ (since $80 > 60$)
4. **Step4:** Finally, we get the longest increasing subsequence which is $[10, 22, 33, 50, 60, 80]$ with a length of 6.

Below is the figure that illustrates the above explained steps:

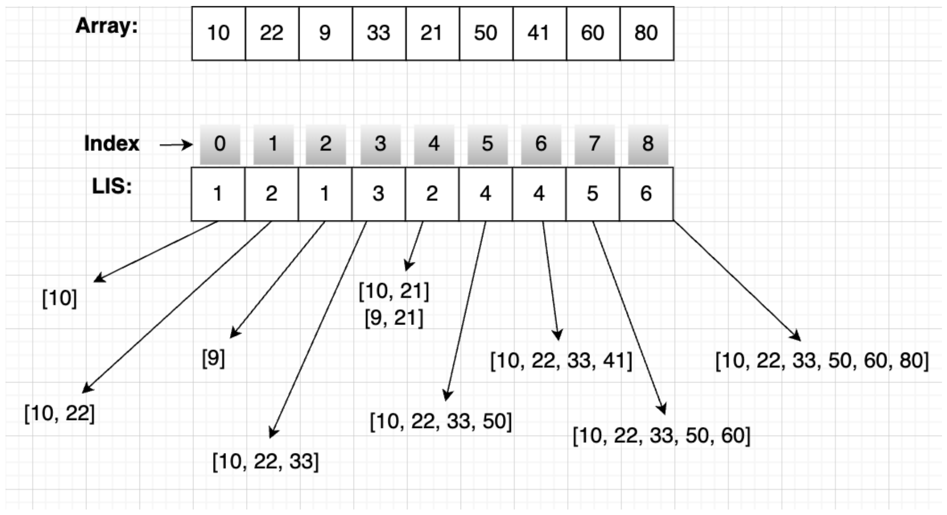


Figure 31: LIS Subsequence

4.6 Coding Problem

Question: Given an integer array nums, return the length and maximum subsequence of the longest strictly increasing subsequence. Below are the test cases to compute the LIS:

1. $[10, 22, 9, 33, 21, 50, 41, 60, 80]$
2. $[10, 9, 2, 5, 3, 7, 101, 18]$
3. $[8, -6, 13, 1, 20, 2, 5, -9]$
4. $[5, 5, 5, 5, 5, 5]$
5. $[8, 7, 6, 5, 4]$
6. $[4, 5, 6, 7, 8, 9]$

Solution:

Please refer the 'LIS.py' file for complete solution.

Reflection:

During my synthesis assignments, I delved into several essential concepts, including:

1. Graph Theory: Understanding graphs, their properties, terminologies, and graph representations.
2. Depth-First Search (DFS): Exploring graph structures using DFS, which involves visiting nodes in a specific order with 3 different strategies.
3. Dijkstra's Algorithm: Learning about this shortest path algorithm, which efficiently finds the optimal path between nodes in a weighted graph.
4. Longest Increasing Subsequence (LIS): Solving problems related to finding the longest increasing subsequence within a given sequence.

Additionally, I explored practical aspects such as implementing adjacency matrices and adjacency lists for graph representation. Dynamic programming also played a crucial role in solidifying my understanding. Reflecting on this experience, I appreciate how these concepts interconnect and contribute to problem-solving in various domains. The hands-on coding problems further reinforced my knowledge and sharpened my skills.

Acknowledgements:

I would like to express my sincere gratitude to the following individuals and resources for their invaluable contributions to this assignment:
1) Prof. Bruce Maxwell: Thank you, Professor Bruce Maxwell for your guidance in this synthesis assignment. Your expertise in algorithms greatly influenced my work.
2) Classmates and TA's: I appreciate the discussions of my classmates, and TA's who clarified my doubts.
3) DPV Algorithms Textbook: This textbook was a useful resource for me to understand the basics of algorithms.
4) https://brilliant.org/wiki/depth-first-search-dfs/: This website provided explanations for DFS.
5) https://www.programiz.com/dsa/dijkstra-algorithm: This website was useful to quickly understand the Dijkstra's algorithm.
6) https://www.geeksforgeeks.org/longest-increasing-subsequence-dp-3/: This website was helpful to understand Longest Increasing Subsequence Problem.
7) https://medium.com/basecs/a-gentle-introduction-to-graph-theory-77969829ead8: This was a useful resource to learn Graph Theory.