# Graph Data Structures and Algorithms in Python

## 1. Introduction to Graph Data Structures

Graphs stand as pivotal non-linear data structures in computer science, adept at modeling intricate relationships between distinct entities. Composed of vertices, also known as nodes, and edges that establish connections between these vertices, graphs offer a versatile framework for representing a wide array of real-world scenarios.[2] Their capacity to abstract complex systems makes them invaluable in fields ranging from social network analysis and transportation logistics to computer networking and beyond.[2] A fundamental understanding of graph concepts empowers individuals to tackle multifaceted problems, including the optimization of network designs, the planning of efficient routes, and the analysis of interconnected data.[2] This report aims to provide a comprehensive exploration of essential graph terminology, prevalent methods for representing graphs in Python, the concept of path matrices, fundamental graph traversal algorithms such as Breadth-First Search (BFS) and Depth-First Search (DFS), and critical Minimum Spanning Tree (MST) algorithms, namely Kruskal's and Prim's algorithms, all accompanied by illustrative Python implementations, as requested.

## 2. Fundamental Graph Terminology

At its core, a graph *G* is mathematically defined as a non-empty set of vertices *V* and a set of edges *E*, where each edge serves to connect a pair of these vertices.[2] This relationship can be formally expressed as *G = (V, E)*. Within this structure, a **vertex**, often interchangeably termed a **node**, represents a fundamental entity within the graph.[2] In practical applications, these vertices can embody diverse elements, such as individuals within a social network or intersections along a transportation route.[2] An **edge**, conversely, signifies a connection or relationship between two vertices.[2] These connections can be either directed or undirected, depending on the nature of the relationship being modeled.

In a **directed graph**, also known as a digraph, each edge possesses a specific direction, thereby indicating a one-way relationship from one vertex to another.[2] This directionality is crucial in representing scenarios where the flow of information or the relationship itself is not reciprocal, such as follower relationships on social media platforms or the hyperlinks connecting web pages.[2] Conversely, an **undirected graph** features edges that lack a specific direction, signifying a bidirectional connection between the connected vertices.[2] This type of graph is suitable for modeling symmetric relationships, such as friendships in a social network or roads in a city

where travel is possible in both directions.[2]

Graphs can also be categorized based on the presence or absence of weights associated with their edges. A **weighted graph** assigns a numerical value, or weight, to each edge, which can represent various properties such as the cost of traversing the edge, the distance between the connected vertices, or the capacity of the connection.[2] Road networks with varying distances between cities or airline routes with different flight durations serve as prime examples of weighted graphs.[2] In contrast, an **unweighted graph** does not have any weights associated with its edges; it solely focuses on the connectivity between the nodes.[2] A simple network illustrating connections without any associated cost would be an example of an unweighted graph.

The **degree of a vertex** in a graph refers to the number of edges that are incident to that vertex. In directed graphs, this concept is further refined into the **in-degree**, which counts the number of edges pointing towards the vertex, and the **out-degree**, which counts the number of edges originating from the vertex.[2] A **path** in a graph is defined as a sequence of vertices where each consecutive pair of vertices is connected by an edge.[3] A **cycle** is a special type of path that begins and ends at the same vertex.[7] A graph is considered **connected** if there exists a path between every pair of vertices within the graph.[2] Conversely, a **disconnected graph** comprises separate components, where no path exists between vertices belonging to different components.[2] Finally, a **tree** is a specific type of connected graph that contains no cycles.[10]

The distinction between directed and undirected graphs, as well as the presence or absence of weights, allows for a nuanced representation of relationships and their attributes. This classification is fundamental as it dictates the types of algorithms and analyses that can be effectively applied to a given graph model.

### 3. Representing Graphs in Python

To effectively utilize graphs in computational tasks, they must be represented in computer memory. Two of the most prevalent methods for achieving this are through the use of adjacency matrices and adjacency lists.[7]

#### 3.1. Adjacency Matrix

An **adjacency matrix** provides a way to represent a graph using a square matrix, or a two-dimensional array, of size $V \times V$, where $V$ denotes the number of vertices in the graph.[1] For unweighted graphs, the entry at matrix[i][j] is typically set to 1 if an edge

exists between vertex *i* and vertex *j*, and 0 otherwise.[1] In the case of weighted graphs, this entry would store the weight of the edge connecting the two vertices, with 0 or a designated value like infinity indicating the absence of an edge.[1] A key characteristic of adjacency matrices for undirected graphs is their symmetry; since an edge between *i* and *j* implies an edge between *j* and *i*, the value at matrix[i][j] will be the same as the value at matrix[j][i].[13]

In Python, an adjacency matrix for an undirected graph can be implemented using a list of lists, as demonstrated below [12]:

Python

```
def create_adjacency_matrix(V, edges):
    matrix = [ * V for _ in range(V)]
    for u, v in edges:
        matrix[u][v] = 1
        matrix[v][u] = 1  # For undirected graph
    return matrix
```

Furthermore, the networkx library in Python offers a convenient way to generate an adjacency matrix from a graph object [16]:

Python

```
import networkx as nx
G = nx.Graph([(0, 1), (1, 2), (2, 0)])
adj_matrix = nx.adjacency_matrix(G).toarray()
print(adj_matrix)
```

Adjacency matrices are particularly well-suited for representing **dense graphs**, where the number of edges is significant relative to the number of vertices.[13] One of their primary advantages is the constant time complexity, O(1), for checking whether an edge exists between any two given vertices.[14] However, a notable drawback is their space complexity of O(*V*^2), which remains constant regardless of the actual number of edges present in the graph.[12] This can lead to inefficient memory usage for **sparse**

**graphs**, where the number of edges is considerably smaller than the potential maximum.

### 3.2. Adjacency List

An **adjacency list** offers an alternative approach to graph representation, where each vertex in the graph maintains a list of its adjacent vertices.[1] This is often implemented using a dictionary where the keys are the vertices, and the values are lists containing their respective neighbors.[19] For weighted graphs, the list associated with each vertex can store pairs, with each pair containing a neighbor and the weight of the edge connecting them.[7]

In Python, an adjacency list for an undirected graph can be implemented using a standard dictionary [19]:

Python

```
def create_adjacency_list(V, edges):
    adj_list = {i: for i in range(V)}
    for u, v in edges:
        adj_list[u].append(v)
        adj_list[v].append(u)  # For undirected graph
    return adj_list
```

Alternatively, the collections.defaultdict in Python provides a convenient way to create an adjacency list, automatically handling the creation of a list for a new vertex when an edge involving it is added [19]:

Python

```
from collections import defaultdict
def create_adjacency_list_defaultdict(V, edges):
    adj_list = defaultdict(list)
    for u, v in edges:
        adj_list[u].append(v)
```

```
    adj_list[v].append(u)  # For undirected graph
return adj_list
```

Adjacency lists are particularly advantageous for representing **sparse graphs** because they only store the edges that actually exist in the graph, leading to more efficient memory usage.[12] The space complexity of an adjacency list is O($V$ + $E$), where $V$ is the number of vertices and $E$ is the number of edges.[7] Retrieving the neighbors of a specific vertex is efficient, taking O(degree(vertex)) time. However, checking for the existence of a particular edge between two vertices might require iterating through the neighbor list of one of the vertices, which can be less direct than the O(1) lookup in an adjacency matrix.

The choice between an adjacency matrix and an adjacency list hinges primarily on the density of the graph and the specific operations that need to be performed frequently. For graphs with a high number of edges, the adjacency matrix offers quick edge lookups, while for graphs with relatively few edges, the adjacency list provides a more memory-efficient representation.

**Table 1: Comparison of Graph Representations**

| Feature | Adjacency Matrix | Adjacency List |
|---|---|---|
| Representation | 2D array ($V$ x $V$) | Dictionary/List of neighbors for each vertex |
| Space Complexity | O($V$^2) | O($V$ + $E$) |
| Edge Existence | O(1) | O(degree($u$)) in the worst case |
| Neighbor Retrieval | O($V$) | O(degree($u$)) |
| Best for | Dense graphs | Sparse graphs |
| Implementation | Relatively straightforward for simple graphs | Can be more complex for weighted/directed graphs |

### 4. The Concept of a Path Matrix

In the realm of graph theory, a **path matrix** serves as a specialized square matrix

designed to represent the existence of paths between any two nodes within a graph.[25] For a graph with *n* vertices, the path matrix is an *n* x *n* binary matrix where the entry at path_matrix[i][j] is 1 if there is a path from vertex *i* to vertex *j*, whether direct or indirect, and 0 otherwise.[26]

The primary **purpose** of a path matrix is to enable a quick determination of whether a route exists between any two nodes in a network.[26] This representation proves particularly useful in applications such as mapping systems and connectivity analysis.[25] For instance, in a path matrix representing train routes, one can readily ascertain if a connection exists between two cities.

The **construction** of a path matrix can be achieved through various methods. One approach involves deriving it from the adjacency matrix by considering paths of different lengths within the graph.[26] However, the **Warshall algorithm** stands out as a commonly employed and efficient method for computing the path matrix.[25] This algorithm operates by iteratively considering each vertex in the graph as a potential intermediate point in the paths between all other pairs of vertices. The process begins by initializing the path matrix with the adjacency matrix of the graph. Then, for each vertex *k* from 0 to *n*-1, the algorithm updates the entry path_matrix[i][j] for all pairs of vertices (*i*, *j*) such that if there is a path from *i* to *k* and a path from *k* to *j*, then there must also be a path from *i* to *j*.[26] This transitive closure is captured by setting path_matrix[i][j] to 1 if path_matrix[i][k] is 1 and path_matrix[k][j] is 1.

A conceptual Python implementation of the Warshall algorithm is as follows:

Python

```
def warshall_algorithm(adj_matrix):
    n = len(adj_matrix)
    path_matrix = [row[:] for row in adj_matrix] # Initialize with adjacency matrix
    for k in range(n):
        for i in range(n):
            for j in range(n):
                if path_matrix[i][k] and path_matrix[k][j]:
                    path_matrix[i][j] = 1
    return path_matrix
```

The path matrix can also be instrumental in **checking graph connectivity**.[29] By analyzing the entries of the path matrix, one can determine if a directed graph is strongly, unilaterally, or weakly connected. A directed graph is considered **strongly connected** if there is a path between every pair of vertices in both directions, which is reflected in a path matrix where all entries are 1.[29] A graph is **unilaterally connected** if, for every pair of vertices ($u$, $v$), there exists a directed path from $u$ to $v$ OR a directed path from $v$ to $u$ (or both).[29] This might be indicated by the upper or lower triangle of the path matrix (excluding the diagonal) containing all 1s. Finally, a directed graph is **weakly connected** if there is a path between every two vertices in the underlying undirected graph (formed by ignoring the direction of the edges).[29] In terms of the directed path matrix, if the graph is neither strongly nor unilaterally connected, it is classified as weakly connected.

The path matrix offers a compact and effective way to represent reachability within a graph. The Warshall algorithm provides an efficient means to compute this matrix, enabling swift checks for path existence and different forms of graph connectivity.

## 5. Graph Traversal Algorithms in Python

Graph traversal algorithms are essential tools for systematically visiting all the vertices within a graph.[4] Two fundamental traversal algorithms are Breadth-First Search (BFS) and Depth-First Search (DFS), each employing a distinct strategy for exploring the graph.

### 5.1. Breadth-First Search (BFS)

**Breadth-First Search (BFS)** is a graph traversal algorithm that explores a graph level by level, beginning from a specified source node.[30] It starts by visiting all the immediate neighbors of the source node before moving on to the next level of neighbors, and this process continues until all reachable vertices have been visited.[30] BFS relies on a **queue** data structure, operating on a First-In, First-Out (FIFO) principle, to keep track of the nodes that need to be visited.[30] A crucial property of BFS is its ability to guarantee finding the shortest path between nodes in unweighted graphs, as it explores neighbors level by level.[30]

A Python implementation of BFS using an adjacency list representation of the graph is as follows [31]:

Python

```
from collections import deque

def bfs(graph, start_node):
    visited = set()
    queue = deque([start_node])
    visited.add(start_node)
    result =
    while queue:
        node = queue.popleft()
        result.append(node)
        for neighbor in graph.get(node,):
            if neighbor not in visited:
                visited.add(neighbor)
                queue.append(neighbor)
    return result
```

BFS finds applications in various domains, including determining the shortest path in unweighted graphs, network routing algorithms, and exploring connections in social networks.[30] The time complexity of BFS is O($V$ + $E$), where $V$ is the number of vertices and $E$ is the number of edges in the graph.[30] The space complexity is O($V$) due to the memory required to store the visited set and the queue.[30]

The level-by-level exploration inherent in BFS makes it particularly suitable for scenarios where the shortest path in terms of the number of edges is sought, or when all nodes within a certain distance from a starting node need to be examined.

### 5.2. Depth-First Search (DFS)

**Depth-First Search (DFS)** is another fundamental graph traversal algorithm that explores as far as possible along each branch of the graph before backtracking to explore alternative branches.[36] DFS can be implemented using either a recursive approach or an iterative approach that employs a stack (Last-In, First-Out or LIFO) data structure.[36]

A recursive Python implementation of DFS is shown below [36]:

Python

```
def dfs_recursive(graph, node, visited):
    visited.add(node)
    print(node, end=" ")
    for neighbor in graph.get(node,):
        if neighbor not in visited:
            dfs_recursive(graph, neighbor, visited)

def dfs(graph, start_node):
    visited = set()
    dfs_recursive(graph, start_node, visited)
```

An iterative Python implementation of DFS using a stack is as follows [38]:

```
Python


def dfs_iterative(graph, start_node):
    visited = set()
    stack = [start_node]
    result =
    while stack:
        node = stack.pop()
        if node not in visited:
            visited.add(node)
            result.append(node)
            neighbors = graph.get(node,)
            for neighbor in reversed(neighbors): # Process neighbors in a consistent order
                stack.append(neighbor)
    return result
```

DFS has a wide range of applications, including pathfinding algorithms, detecting cycles within a graph, performing topological sorting on directed acyclic graphs, and exploring all the connected components of a graph.[36] Similar to BFS, the time complexity of DFS is O($V$ + $E$), and the space complexity is O($V$).[36]

The depth-first exploration strategy of DFS makes it well-suited for tasks that require finding a specific path between two nodes or identifying the presence of cycles. The

choice between the recursive and iterative implementations often depends on the specific requirements of the problem and considerations such as the potential for stack overflow in very deep graphs when using recursion.

**6. Minimum Spanning Trees**

A **Minimum Spanning Tree (MST)** is a fundamental concept in graph theory, particularly relevant for connected, weighted, and undirected graphs. An MST is a subgraph that forms a tree, connects all the vertices of the original graph, and has the minimum possible total weight of its edges.[8] A key property of an MST for a graph with $V$ vertices is that it contains exactly $V$-1 edges [45] and is devoid of any cycles.[10] It's worth noting that if a graph contains edges with the same weight, the MST might not be unique [43]; however, if all edge weights are distinct, the MST will be unique.[43]

The significance of finding a Minimum Spanning Tree lies in its ability to identify the most cost-effective way to connect all nodes in a network.[43] This has profound implications in various real-world applications, such as designing efficient communication networks, optimizing logistics and transportation routes, and performing cluster analysis in data mining.[10] The core idea is to ensure that all entities are connected with the minimum possible total cost or resource usage.

**7. Kruskal's Algorithm for MST (with Python Implementation)**

**Kruskal's algorithm** is a greedy algorithm that provides an effective method for finding the Minimum Spanning Tree of a connected, weighted, and undirected graph.[8] The algorithm operates by iteratively adding the cheapest edges to the MST, provided that these additions do not create a cycle.

The steps of Kruskal's algorithm are as follows:

1. Begin by sorting all the edges in the graph in non-decreasing order based on their weights.[10]
2. Initialize an empty set to represent the Minimum Spanning Tree.
3. Iterate through the sorted edges. For each edge ($u$, $v$) with weight $w$:
    - Check if adding this edge to the MST would create a cycle.[10] This check is typically performed using a **Disjoint Set Union (DSU)** data structure, which helps in keeping track of the connected components.
    - If adding the edge does not result in a cycle, add the edge to the MST and perform a union operation on the sets containing vertices $u$ and $v$ in the DSU.[10]
4. Repeat step 3 until the MST contains $V$-1 edges, where $V$ is the number of vertices

in the graph.[10]

A Python implementation of Kruskal's algorithm is provided below [60]:

Python

```
class DSU:
    def __init__(self, vertices):
        self.parent = list(range(vertices))
        self.rank =  * vertices

    def find(self, i):
        if self.parent[i] == i:
            return i
        self.parent[i] = self.find(self.parent[i])
        return self.parent[i]

    def union(self, x, y):
        root_x = self.find(x)
        root_y = self.find(y)
        if root_x!= root_y:
            if self.rank[root_x] < self.rank[root_y]:
                self.parent[root_x] = root_y
            elif self.rank[root_x] > self.rank[root_y]:
                self.parent[root_y] = root_x
            else:
                self.parent[root_y] = root_x
                self.rank[root_x] += 1
            return True
        return False

def kruskal(num_vertices, edges):
    edges.sort(key=lambda x: x[1]) # Sort by weight
    mst =
    dsu = DSU(num_vertices)
    for u, v, weight in edges:
        if dsu.union(u, v):
```

```
            mst.append((u, v, weight))
    return mst
```

The time complexity of Kruskal's algorithm is typically O($E$ log $E$) or O($E$ log $V$) due to the initial sorting of the edges.[52] The space complexity is O($E$ + $V$) to accommodate the storage of the edges and the DSU structure.[47] Kruskal's algorithm is particularly efficient for **sparse graphs**, where the number of edges is significantly less than the square of the number of vertices.[44] The edge-centric nature of Kruskal's algorithm, coupled with the efficient cycle detection provided by the DSU, makes it a valuable tool for constructing Minimum Spanning Trees, especially in graphs with a relatively low edge density.

**8. Prim's Algorithm for MST (with Python Implementation)**

**Prim's algorithm** offers another greedy approach to finding the Minimum Spanning Tree of a connected, weighted, and undirected graph.[8] Unlike Kruskal's, Prim's algorithm builds the MST one vertex at a time, starting from an arbitrary root vertex.

The steps involved in Prim's algorithm are as follows:

1. Select an arbitrary vertex to serve as the starting point for the MST and add it to the set of vertices in the MST.
2. Initialize a set of candidate edges as all the edges that connect a vertex in the MST to a vertex that is not yet in the MST.
3. While there are vertices in the graph that are not yet included in the MST:
    - Select the edge from the candidate set that has the minimum weight.
    - Add the vertex at the other end of this edge (the one not currently in the MST) to the MST set.
    - Add the selected edge to the MST.
    - Update the set of candidate edges by adding any new edges that connect the newly added vertex to vertices that are still not in the MST.

A Python implementation of Prim's algorithm using the heapq module for a priority queue is provided below [77]:

```
Python
```

```
import heapq
```

```
def prim(graph, start_node):
    mst =
    visited = {start_node}
    edges = [(weight, start_node, neighbor) for neighbor, weight in
graph.get(start_node, {}).items()]
    heapq.heapify(edges)

    while edges:
        weight, u, v = heapq.heappop(edges)
        if v not in visited:
            visited.add(v)
            mst.append((u, v, weight))
            for neighbor, weight in graph.get(v, {}).items():
                if neighbor not in visited:
                    heapq.heappush(edges, (weight, v, neighbor))
    return mst
```

The time complexity of Prim's algorithm using a binary heap (as implemented with heapq) is O((*V* + *E*) log *V*).[50] This can be further improved to O(*E* + *V* log *V*) by using more advanced data structures like Fibonacci heaps.[50] The space complexity is O(*V* + *E*) to store the graph and the priority queue.[50] Prim's algorithm is generally more efficient for **dense graphs**, where the number of edges is high.[44] The vertex-centric approach of Prim's algorithm, utilizing a priority queue to efficiently select the minimum weight edge connecting to the growing MST, makes it a strong contender for graphs with a high edge density.

### 9. Comparing Kruskal's and Prim's Algorithms

Both Kruskal's and Prim's algorithms stand as prominent greedy algorithms for solving the problem of finding the Minimum Spanning Tree in a connected, weighted, undirected graph.[45] Despite their shared objective, they employ distinct strategies in their pursuit of the MST.

In terms of **approach**, Kruskal's algorithm is fundamentally edge-based. It begins by considering all the edges in the graph and iteratively adds the edges with the smallest weights to the MST, ensuring that no cycles are formed.[67] This process can be visualized as building a forest of trees, which gradually merge together until a single MST is obtained.[48] Conversely, Prim's algorithm adopts a vertex-based approach. It starts with an arbitrary single vertex and progressively grows the MST by adding the

minimum weight edge that connects the current tree to an unvisited vertex.[67] This method ensures that the MST remains a single connected component throughout the algorithm's execution.[67]

Regarding **efficiency**, Kruskal's algorithm typically exhibits a time complexity of O($E$ log $V$) or O($E$ log $E$) [52], making it particularly well-suited for **sparse graphs**.[44] Prim's algorithm, when implemented with a binary heap, has a time complexity of O(( $V$ + $E$) log $V$), which can be improved to O($E$ + $V$ log $V$) using Fibonacci heaps.[50] This makes Prim's algorithm generally more efficient for **dense graphs**.[44]

The choice of **data structures** also differentiates the two algorithms. Kruskal's algorithm relies on the **Disjoint Set Union (DSU)** data structure to efficiently detect cycles when adding edges.[45] In contrast, Prim's algorithm commonly employs a **priority queue** (often implemented as a min-heap) to quickly find the minimum weight edge that connects the growing MST to an unvisited vertex.[10]

In terms of **implementation complexity**, Kruskal's algorithm might be considered simpler to implement for sparse graphs, primarily due to the straightforward sorting of edges and the use of the DSU.[67] Prim's algorithm, especially when optimized with a priority queue, can be more involved to implement, although it might be simpler in dense graphs where the adjacency matrix representation is often used.[67]

Finally, regarding their ability to handle different graph types, Kruskal's algorithm is more versatile as it can gracefully handle **disconnected graphs**, producing a minimum spanning forest (a collection of MSTs, one for each connected component).[43] Prim's algorithm, on the other hand, is typically designed to work with **connected graphs** to produce a single MST.[43]

**Table 2: Comparison of Kruskal's and Prim's Algorithms**

| Feature | Kruskal's Algorithm | Prim's Algorithm |
|---|---|---|
| Approach | Edge-based (sorts all edges) | Vertex-based (grows from a single vertex) |
| Cycle Detection | Explicit (using DSU) | Implicit (only connects to unvisited vertices) |
| Time Complexity | O($E$ log $E$) or O($E$ log $V$) | O(( $V$ + $E$) log $V$) or O($E$ + $V$ log $V$) |

| | | |
|---|---|---|
| Space Complexity | O($E + V$) | O($V + E$) |
| Best for | Sparse graphs | Dense graphs |
| Starting Point | Starts with all edges, adds cheapest non-cyclic | Starts with a single vertex, adds cheapest edge to it |
| Connectivity | Can handle disconnected graphs | Typically for connected graphs |

The choice between Kruskal's and Prim's algorithm often boils down to the density of the graph being analyzed. For graphs with a relatively small number of edges, Kruskal's algorithm's efficiency in sorting edges makes it a preferred choice. Conversely, for graphs with a large number of edges, Prim's algorithm's focus on vertices and its efficient use of a priority queue tend to yield better performance.

**10. Conclusion**

Graphs are a remarkably powerful and flexible data structure, providing a means to model a vast array of relationships found in the real world. A solid understanding of fundamental graph terminology, including the distinctions between directed and undirected graphs, as well as weighted and unweighted graphs, is essential for effectively working with these structures. The choice of graph representation, whether an adjacency matrix or an adjacency list, significantly impacts the efficiency of various graph operations, with adjacency matrices favoring dense graphs and adjacency lists being more suitable for sparse graphs.

Graph traversal algorithms, such as Breadth-First Search (BFS) and Depth-First Search (DFS), offer systematic ways to explore the vertices and edges of a graph, each with its own unique characteristics and applications. BFS excels in finding shortest paths in unweighted graphs, while DFS is adept at exploring the depth of a graph and detecting cycles.

Minimum Spanning Trees (MSTs) provide a critical solution for connecting all vertices in a weighted graph with the minimum total edge weight. Both Kruskal's and Prim's algorithms are effective greedy strategies for finding MSTs, with their performance characteristics varying based on the density of the graph. Kruskal's algorithm, with its edge-based approach and use of the Disjoint Set Union data structure, is often preferred for sparse graphs. In contrast, Prim's algorithm, which adopts a vertex-based approach and typically utilizes a priority queue, tends to perform better

on dense graphs. Python, with its rich set of built-in data structures and the availability of libraries like networkx and heapq, provides a convenient and powerful environment for implementing and working with these fundamental graph concepts and algorithms.

**Works cited**


1. Graph in Data Structure | Types & Explanation - Simplilearn.com, accessed April 20, 2025, https://www.simplilearn.com/tutorials/data-structure-tutorial/graphs-in-data-structure
2. Graph terminology in data structure | GeeksforGeeks, accessed April 20, 2025, https://www.geeksforgeeks.org/graph-terminology-in-data-structure/
3. Graph - Explore - LeetCode, accessed April 20, 2025, https://leetcode.com/explore/learn/card/graph/
4. Introduction to Graph Data Structure | GeeksforGeeks, accessed April 20, 2025, https://www.geeksforgeeks.org/introduction-to-graphs-data-structure-and-algorithm-tutorials/
5. Graphs and Networks for Beginners - Hypermode, accessed April 20, 2025, https://hypermode.com/blog/graphs-and-networks
6. Graph (discrete mathematics) - Wikipedia, accessed April 20, 2025, https://en.wikipedia.org/wiki/Graph_(discrete_mathematics)
7. Graphs - Terminology and Representation, accessed April 20, 2025, https://sites.radford.edu/~nokie/classes/360/graphs-terms.html
8. Graph Data Structure: Directed, Acyclic, etc - Interview Cake, accessed April 20, 2025, https://www.interviewcake.com/concept/java/graph
9. Graph Terminology and Properties | Data Structures Class Notes - Fiveable, accessed April 20, 2025, https://library.fiveable.me/data-structures/unit-10/graph-terminology-properties/study-guide/07czPf6ALzmjmnV6
10. What is Minimum Spanning Tree (MST) | GeeksforGeeks, accessed April 20, 2025, https://www.geeksforgeeks.org/what-is-minimum-spanning-tree-mst/
11. Adjacency Matrix in Python | GeeksforGeeks, accessed April 20, 2025, https://www.geeksforgeeks.org/adjacency-matrix-in-python/
12. Building and Analyzing Graphs with the Adjacency Matrix | CodeSignal Learn, accessed April 20, 2025, https://codesignal.com/learn/courses/mastering-graphs-in-python/lessons/building-and-analyzing-graphs-with-the-adjacency-matrix?ref=toolpasta.com
13. Adjacency Matrix Representation | GeeksforGeeks, accessed April 20, 2025, https://www.geeksforgeeks.org/adjacency-matrix/
14. Graphy Adjacency Matrix (Full Guide With Code Implementation) - WsCube Tech, accessed April 20, 2025, https://www.wscubetech.com/resources/dsa/graph-adjacency-matrix
15. Representing Graph using Adjacency Matrix - Youcademy, accessed April 20, 2025, https://youcademy.org/graph-with-adjacency-matrix/

16. adjacency_matrix — NetworkX 3.4.2 documentation, accessed April 20, 2025, https://networkx.org/documentation/stable/reference/generated/networkx.linalg.graphmatrix.adjacency_matrix.html
17. How do I generate an adjacency matrix of a graph from a dictionary in python?, accessed April 20, 2025, https://stackoverflow.com/questions/37353759/how-do-i-generate-an-adjacency-matrix-of-a-graph-from-a-dictionary-in-python
18. Best way to represent Graph data structure in Python - Reddit, accessed April 20, 2025, https://www.reddit.com/r/Python/comments/64vm47/best_way_to_represent_graph_data_structure_in/
19. Adjacency List in Python | GeeksforGeeks, accessed April 20, 2025, https://www.geeksforgeeks.org/adjacency-list-in-python/
20. Adjacency List (With Code in C, C++, Java and Python) - Programiz, accessed April 20, 2025, https://www.programiz.com/dsa/graph-adjacency-list
21. Representing a Graph, accessed April 20, 2025, https://bradfieldcs.com/algos/graphs/representing-a-graph/
22. Adjacency List Representation | GeeksforGeeks, accessed April 20, 2025, https://www.geeksforgeeks.org/adjacency-list-meaning-definition-in-dsa/
23. Python - How to convert a List into an Adjacency List for graphs structure - Stack Overflow, accessed April 20, 2025, https://stackoverflow.com/questions/75113537/python-how-to-convert-a-list-into-an-adjacency-list-for-graphs-structure
24. Difficulty understanding Adjacency List representation of graph in Python - Reddit, accessed April 20, 2025, https://www.reddit.com/r/learnpython/comments/in4zqd/difficulty_understanding_adjacency_list/
25. www.scribd.com, accessed April 20, 2025, https://www.scribd.com/document/608814655/Path-matrix#:~:text=and%20graph%20theory.-,It%20defines%20a%20path%20matrix%20as%20a%20matrix%20that%20represents,path%20matrices%20include%20mapping%20systems.
26. Path Matrix | PDF | Teaching Methods & Materials - Scribd, accessed April 20, 2025, https://www.scribd.com/document/608814655/Path-matrix
27. 7-8. path matrix, accessed April 20, 2025, https://udrc.lkouniv.ac.in/Content/DepartmentContent/SM_a71ab2a8-22e2-4b41-80ad-08c7fd91e51e_58.pdf
28. What does it mean by "Path Matrix" and "Transitive Closure" of a graph (Directed and Undirected)? - Stack Overflow, accessed April 20, 2025, https://stackoverflow.com/questions/6757031/what-does-it-mean-by-path-matrix-and-transitive-closure-of-a-graph-directed
29. Check if a graph is Strongly, Unilaterally or Weakly connected ..., accessed April 20, 2025, https://www.geeksforgeeks.org/check-if-a-graph-is-strongly-unilaterally-or-weakly-connected/
30. Breadth-First Search in Python: A Guide with Examples | DataCamp, accessed

April 20, 2025, https://www.datacamp.com/tutorial/breadth-first-search-in-python
31. Breadth First Search or BFS for a Graph in Python | GeeksforGeeks, accessed April 20, 2025, https://www.geeksforgeeks.org/python-program-for-breadth-first-search-or-bfs-for-a-graph/
32. Breadth First Search or BFS for a Graph | GeeksforGeeks, accessed April 20, 2025, https://www.geeksforgeeks.org/breadth-first-search-or-bfs-for-a-graph/
33. BFS Graph Algorithm(With code in C, C++, Java and Python) - Programiz, accessed April 20, 2025, https://www.programiz.com/dsa/graph-bfs
34. Breadth-First Search Algorithm - Kaggle, accessed April 20, 2025, https://www.kaggle.com/code/liamhealy/breadth-first-search-algorithm
35. Breadth-first search (BFS) of BST in Python - Visualization and Code, accessed April 20, 2025, https://csanim.com/tutorials/breadth-first-search-python-visualization-and-code
36. Depth First Search or DFS for a Graph – Python | GeeksforGeeks, accessed April 20, 2025, https://www.geeksforgeeks.org/python-program-for-depth-first-search-or-dfs-for-a-graph/
37. Depth-First Search in Python: Traversing Graphs and Trees ..., accessed April 20, 2025, https://www.datacamp.com/tutorial/depth-first-search-in-python
38. Depth-First Search (DFS) Algorithm in Python: A Comprehensive Guide - Aswnss Blog, accessed April 20, 2025, https://aswnss.hashnode.dev/implementing-depth-first-search-dfs-algorithm-in-python-a-comprehensive-guide
39. Implementing Depth-First Search in Python to Traverse a Binary Tree - llego.dev, accessed April 20, 2025, https://llego.dev/posts/implementing-depth-first-search-python-traverse-binary-tree/
40. Depth-First Search (DFS) - CelerData, accessed April 20, 2025, https://celerdata.com/glossary/depth-first-search-dfs
41. Depth First Search (DFS) Algorithm - Programiz, accessed April 20, 2025, https://www.programiz.com/dsa/graph-dfs
42. Depth First Search or DFS for a Graph | GeeksforGeeks, accessed April 20, 2025, https://www.geeksforgeeks.org/depth-first-search-or-dfs-for-a-graph/
43. Minimum spanning tree - Wikipedia, accessed April 20, 2025, https://en.wikipedia.org/wiki/Minimum_spanning_tree
44. Minimum Spanning Tree (MST) in Graph Data Structure - Youcademy, accessed April 20, 2025, https://youcademy.org/minimum-spanning-trees-in-graphs/
45. Minimum spanning tree - (Data Structures) - Vocab, Definition, Explanations | Fiveable, accessed April 20, 2025, https://fiveable.me/key-terms/data-structures/minimum-spanning-tree
46. Minimum spanning trees | Combinatorics Class Notes - Fiveable, accessed April 20, 2025, https://library.fiveable.me/combinatorics/unit-14/minimum-spanning-trees/study-

guide/te5wQ9BCYoD1fH5D
47. Kruskal's Minimum Spanning Tree (MST) Algorithm | GeeksforGeeks, accessed April 20, 2025, https://www.geeksforgeeks.org/kruskals-minimum-spanning-tree-algorithm-greedy-algo-2/
48. 4.3 Minimum Spanning Trees - Algorithms, 4th Edition, accessed April 20, 2025, https://algs4.cs.princeton.edu/43mst/
49. Minimum Spanning Tree (MST) Algorithm in Data Structure - Simplilearn.com, accessed April 20, 2025, https://www.simplilearn.com/tutorials/data-structure-tutorial/minimum-spanning-tree-algorithm-in-data-structure
50. The Minimum Spanning Tree (MST) problem, accessed April 20, 2025, https://www2.seas.gwu.edu/~simhaweb/champalg/mst/mst.html
51. Minimum Spanning Trees, accessed April 20, 2025, https://jeffe.cs.illinois.edu/teaching/algorithms/book/07-mst.pdf
52. Working of Kruskal's Algorithm | Board Infinity, accessed April 20, 2025, https://www.boardinfinity.com/blog/kruskal-algorithm/
53. Kruskal's Algorithm: Key to Minimum Spanning Tree [MST], accessed April 20, 2025, https://www.simplilearn.com/tutorials/data-structure-tutorial/kruskal-algorithm
54. Kruskal's Algorithm - Programiz, accessed April 20, 2025, https://www.programiz.com/dsa/kruskal-algorithm
55. Kruskal's algorithm (Minimum spanning tree) with real-life examples - HackerEarth, accessed April 20, 2025, https://www.hackerearth.com/blog/developers/kruskals-minimum-spanning-tree-algorithm-example/
56. Kruskal's algorithm - Wikipedia, accessed April 20, 2025, https://en.wikipedia.org/wiki/Kruskal%27s_algorithm
57. Kruskal's algorithm in 2 minutes - YouTube, accessed April 20, 2025, https://www.youtube.com/watch?v=71UQH7Pr9kU
58. Kruskal's Minimal Spanning Tree Algorithm - Tutorialspoint, accessed April 20, 2025, https://www.tutorialspoint.com/data_structures_algorithms/kruskals_spanning_tree_algorithm.htm
59. Kruskal's Algorithm - YouTube, accessed April 20, 2025, https://www.youtube.com/watch?v=ivcbalhrcsE
60. Kruskal's Algorithm in Python | GeeksforGeeks, accessed April 20, 2025, https://www.geeksforgeeks.org/kruskals-algorithm-in-python/
61. Implementing Kruskal's and Prim's Algorithms: A Comprehensive Guide - AlgoCademy, accessed April 20, 2025, https://algocademy.com/blog/implementing-kruskals-and-prims-algorithms-a-comprehensive-guide/
62. Kruskal's Algorithm In Python - HeyCoach | Blogs, accessed April 20, 2025, https://blog.heycoach.in/kruskals-algorithm-in-python/
63. arnab132/Kruskals-Graph-Algorithm-Python - GitHub, accessed April 20, 2025,

https://github.com/arnab132/Kruskals-Graph-Algorithm-Python
64. Python - Kruskal's Algorithm for Minimum Spanning Trees - DEV Community, accessed April 20, 2025, https://dev.to/theramoliya/python-kruskals-algorithm-for-minimum-spanning-trees-2bmb
65. Kruskal-Algorithm-in-Python/kruskal.py at master - GitHub, accessed April 20, 2025, https://github.com/nouman-10/Kruskal-Algorithm-in-Python/blob/master/kruskal.py
66. 24- Kruskal Algorithm | Graph Theory | Python - YouTube, accessed April 20, 2025, https://www.youtube.com/watch?v=hupbSXIlSIc
67. Difference between Prim's and Kruskal's algorithm for MST | GeeksforGeeks, accessed April 20, 2025, https://www.geeksforgeeks.org/difference-between-prims-and-kruskals-algorithm-for-mst/
68. Prim's algorithm - Wikipedia, accessed April 20, 2025, https://en.wikipedia.org/wiki/Prim%27s_algorithm
69. Prim's Algorithm - cs.wisc.edu, accessed April 20, 2025, https://pages.cs.wisc.edu/~siena/
70. Prim's Algorithm - Programiz, accessed April 20, 2025, https://www.programiz.com/dsa/prim-algorithm
71. Prim's Algorithm: Example, Time Complexity, Code - WsCube Tech, accessed April 20, 2025, https://www.wscubetech.com/resources/dsa/prim-algorithm
72. Prim's Minimum Spanning Tree (MST) | Simplilearn - Simplilearn.com, accessed April 20, 2025, https://www.simplilearn.com/tutorials/data-structure-tutorial/prims-algorithm
73. Prim's Algorithm for Minimum Spanning Tree (MST) | GeeksforGeeks, accessed April 20, 2025, https://www.geeksforgeeks.org/prims-minimum-spanning-tree-mst-greedy-algo-5/
74. Prim's algorithm in 2 minutes - YouTube, accessed April 20, 2025, https://www.youtube.com/watch?v=cplfcGZmX7I
75. Prim's Algorithm: A Comprehensive Guide to Minimum Spanning Trees – AlgoCademy Blog, accessed April 20, 2025, https://algocademy.com/blog/prims-algorithm-a-comprehensive-guide-to-minimum-spanning-trees/
76. Prim's Spanning Tree Algorithm - Tutorialspoint, accessed April 20, 2025, https://www.tutorialspoint.com/data_structures_algorithms/prims_spanning_tree_algorithm.htm
77. Prim's Algorithm in Python | GeeksforGeeks, accessed April 20, 2025, https://www.geeksforgeeks.org/prims-algorithm-in-python/
78. Graphs in Python - Theory and Implementation - Minimum Spanning Trees - Prim's Algorithm - Stack Abuse, accessed April 20, 2025, https://stackabuse.com/courses/graphs-in-python-theory-and-implementation/lessons/minimum-spanning-trees-prims-algorithm/

79. valandro/python-prim: PRIM algorithm implemented using Python. - GitHub, accessed April 20, 2025, https://github.com/valandro/python-prim
80. minimum spanning tree - Prim's MST Algorithm in Python - Stack Overflow, accessed April 20, 2025, https://stackoverflow.com/questions/58824557/prims-mst-algorithm-in-python
81. Prim's Spanning Tree Algorithm, accessed April 20, 2025, https://bradfieldcs.com/algos/graphs/prims-spanning-tree-algorithm/
82. How to complete this Prim's algorithm? : r/learnpython - Reddit, accessed April 20, 2025, https://www.reddit.com/r/learnpython/comments/px457j/how_to_complete_this_prims_algorithm/
83. Prim's Algorithm - Minimum Spanning Trees in Python - YouTube, accessed April 20, 2025, https://www.youtube.com/watch?v=5ZYBFcPSvvE
84. When should I use Kruskal as opposed to Prim (and vice versa)? - Stack Overflow, accessed April 20, 2025, https://stackoverflow.com/questions/1195872/when-should-i-use-kruskal-as-opposed-to-prim-and-vice-versa
85. Difference Between Prims and Kruskal Algorithm - Shiksha, accessed April 20, 2025, https://www.shiksha.com/online-courses/articles/difference-between-prims-and-kruskal-algorithm-blogId-155863
86. How do Kruskal's and Prim's algorithms compare to each other?, accessed April 20, 2025, https://cs.stackexchange.com/questions/138584/how-do-kruskals-and-prims-algorithms-compare-to-each-other
87. Prim's vs Kruskal's vs Dijkstra's for lowest cost : r/learnprogramming - Reddit, accessed April 20, 2025, https://www.reddit.com/r/learnprogramming/comments/qw5qv3/prims_vs_kruskals_vs_dijkstras_for_lowest_cost/
88. Kruskal's vs Prim's Algorithm | Baeldung on Computer Science, accessed April 20, 2025, https://www.baeldung.com/cs/kruskals-vs-prims-algorithm
89. Difference between Prims and Kruskal Algorithm - BYJU'S, accessed April 20, 2025, https://byjus.com/gate/difference-between-prims-and-kruskal-algorithum/
90. 9.4 Comparison and analysis of Kruskal's and Prim's algorithms - Fiveable, accessed April 20, 2025, https://library.fiveable.me/introduction-algorithms/unit-9/comparison-analysis-kruskals-prims-algorithms/study-guide/0hrf9qkRrglgAGOS