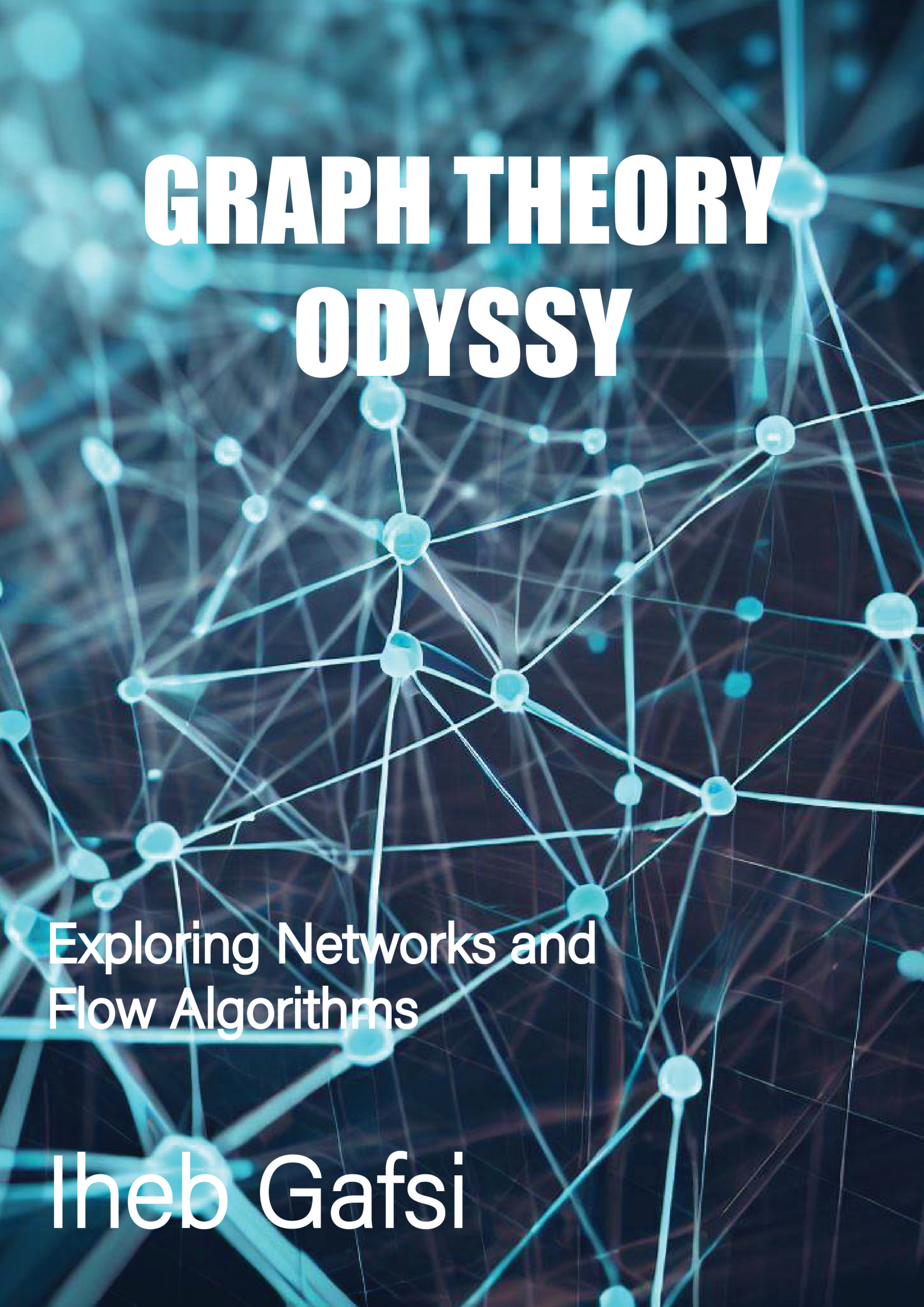

GRAPH THEORY
ODYSSY
Exploring Networks and
Flow Algorithms
Iheb Gafsi

Table of Contents

8- Floyd Warshall FW Algorithm

Learning the Floyd-Warshall algorithm for all pairs shortest paths.

9- Bridges and Articulation Points

Understanding bridges and articulation points in a graph.

10- Tarjan's Algorithm

Exploring Tarjan's algorithm for finding strongly connected components in a graph.

11- Eulerian Paths

Learning about Eulerian paths and circuits in graphs.

12- Traveling Salesman TSP with Dynamic Programming

Understanding the Traveling Salesman Problem and its solution using Dynamic Programming.

13- Ford Fulkerson Network Flow

Exploring the Ford-Fulkerson algorithm for maximum flow in a network.

14- Maximum Cardinality of Bipartite MCBM

Learning about Maximum Cardinality of Bipartite Matching (MCBM) problem and its solutions.

15- Edmonds-Karp Algorithm

Understanding the Edmonds-Karp algorithm for maximum flow in a network.

16- Capacity Scaling Algorithm

Exploring the Capacity Scaling algorithm for maximum flow in a network.

17- Dinic's Algorithm

Learning Dinic's algorithm for maximum flow in a network.

Introduction to Author

Iheb Gafsi, a promising individual, currently pursues his studies in ICT Engineering at the esteemed National Institute of Applied Sciences and Technology of Tunis (INSAT). Although yet to attain his degree, Iheb's passion for technology has already set him on an impressive trajectory. His journey is marked by a profound fascination for the realms of Data Science, Machine Learning, and Android development. Notably, Iheb achieved a significant milestone at the tender age of 16, obtaining an Android Developer Nanodegree from the renowned collaboration between Google and Udacity. This remarkable accomplishment was a direct result of his victory in the Million Arab Coders competition, a testament to his determination and aptitude. Iheb's story serves as a compelling example of youthful ambition and achievement, inspiring others to embark on their own technological endeavors with a sense of purpose and dedication.

Acknowledgments

My journey is a reflection of the incredible support and influence I've received from my parents, friends, and university. Their unwavering dedication has been the driving force behind my achievements. My parents' constant encouragement and belief in my potential have shaped my determination to excel. Alongside them, my friends have provided unwavering support, pushing me to reach higher and strive for greatness. The academic environment at the National Institute of Applied Sciences and Technology of Tunis (INSAT) has been pivotal in honing my skills and nurturing my passion for technology. The rich educational resources and guidance from my university have been instrumental in my growth as a Data Scientist, Machine Learning Enthusiast, and Android developer. I am deeply grateful for the collective impact of my parents, friends, and university, which has not only paved the way for my success but also continues to inspire me to aim higher.

Introduction To Graph Theory

Definition:

Graph Theory is a branch of mathematics and computer science that deals with the study of graphs/networks. Graphs are mathematical structures that represent relationships between pairwise objects that are referred to as vertices or nodes. The connections between these nodes can also be referred to as edges.

Use Cases:

Graph Theory can help us solve several problems, in mathematics, computer science, data science, social sciences, operations research, network analysis, and many more. Graph theory helps in understanding the connections and interactions between different elements in a system, leading to insights, algorithms, and optimizations for various real-world applications.

Types of Graphs:

- **Undirected Graphs:**

  Undirected Graphs are the most generalized form. A graph is an undirected graph when the edges connecting it are undirected so the edge connecting the nodes u and v $(u, v) = (v, u)$. **Figure 1** shows a little example of undirected graphs:

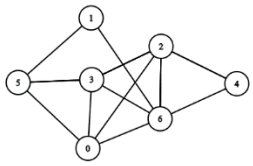


Figure 1: Undirected Graph

- **Directed Graphs:**

  Unlike undirected graphs, directed graphs have directions so the edge connecting the nodes u and v $(u, v) \neq (v, u)$. **Figure 2** illustrates an example of directed graphs:

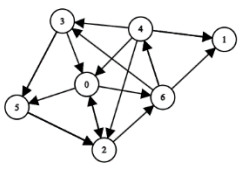


Figure 2: Directed Graph

- **Weighted Graphs:**

  Unlike typical graphs, weighted graphs have weights associated with them so the edge connecting the nodes u and v is represented as (u, v, w) where w is the corresponding weight. **Figure 3** represents an undirected weighted graph:

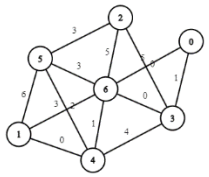


Figure 3: Directed Graph

- **Directed Acyclic Graphs DAGs:**

  Directed Acyclic Graphs (DAGs) are directed graphs without cycles, making them suitable for modeling processes with clear cause-and-effect relationships. On the other hand, directed graphs can have cycles, allowing paths to loop back to the starting node. **Figure 4** shows an example of a DAG:

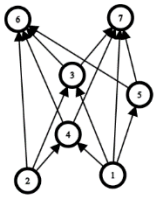


Figure 4: Directed Acyclic Graph

- **Trees:**

  Trees are acyclic, connected graphs with no cycles and a unique starting point called the root. They are widely used in computer science, data structures, and represent hierarchical relationships efficiently. Each node has one parent, except for the root. **Figure 5** shows an instance of a tree:

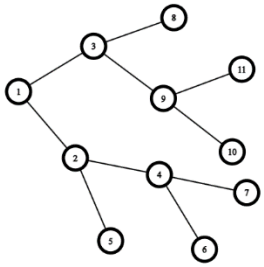


Figure 5: Tree

- **Bipartite Graphs:**

  Bipartite graphs are graphs with two distinct sets of vertices, where edges only connect vertices from different sets. They are used to model relationships between two types of objects or entities and have applications in matching problems, network flow, and social networks. **Figure 6** illustrates what a bipartite graph looks like:

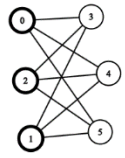


Figure 6: Bipartite Graph

- **Complete Graphs:**

  A complete graph is simply a completely connected graph. Figure 7 below shows an example of a complete graph:

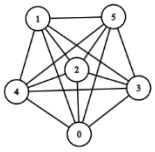


Figure 7: Complete Graph

Representing a Graph:

- **Adjacency Matrix:**

  An adjacency matrix is a tool that allows us to best represent a directed weighted graph. Every element a_{ij} represents the weight of the edge connecting the i'th node to the j'th one. Here's an example illustrated in **Figure 8**:

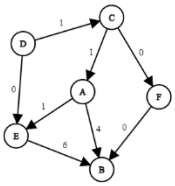


Figure 8: Graph for adjacency matrix

$$Adj = \begin{array}{c} \\ A \\ B \\ C \\ D \\ E \\ F \end{array} \begin{array}{c} \begin{array}{cccccc} A & B & C & D & E & F \end{array} \\ \begin{pmatrix} 0 & 4 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 6 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix} \end{array}$$

- **Adjacency List:**

  An Adjacency List is a way to represent a graph as a map from nodes to lists of edges. Here's the example for the case of **Figure 8**:

```
A →[(B, 4), (E, 1)]
B →[]
C →[(A, 1), (F, 0)]
D →[(C, 1), (E, 0)]
E →[(B, 6)]
F →[(B, 0)]
```

- **Edges List:**

  Unlike other representations we can simply represent our graph with its unordered list of connections. We will use the example of the case of **Figure 8** again:

```
Graph = [(A, B, 4), (A, E, 1), (C, A, 1), (C, F, 0), (D, C, 1),
(D, E, 0), (E, B, 6), (F, B, 0)]
```

Depth-First Search DFS

Definition:

The depth-first search DFS is an algorithm used for traversing or searching a graph data structure, the “depth” in the DFS word refers to the order in which the nodes are explored. The DFS algorithm explores as far as possible along each branch before backtracking until all the nodes that are connected are covered.

Use cases:

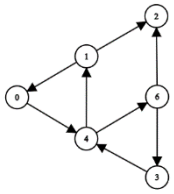
DFS has several significant use cases in graph-related applications. It is commonly employed for pathfinding between nodes, identifying connected components, and performing topological sorting in directed acyclic graphs. Additionally, DFS is valuable in detecting cycles within graphs, solving puzzles, generating mazes, and aiding in decision-making processes for games and AI algorithms. Its versatility and efficiency make it a fundamental tool for a wide range of graph traversal and exploration tasks.

Algorithm:

```
# Variables:
n = number of nodes Example:
graph = the adjacency list representing our graph
visited = [ False, False, …, False]

# The DFS algorithm
def dfs(i):
    if visited[i]:
        return
    visited[i] = True
    for n in graph[i]:
        dfs(n)

# Start from the first node
dfs(0)

```

Example:

Here’s a small example illustrating an example of input outputs for the DFS:



We will use the Python code down below to outline the output of the algorithm on this graph:

```
graph = [ [4], [0, 2], [], [4], [1, 6], [], [2, 3] ]
n = len(graph)
visited = [False]*n

def dfs(i):
    print(i, end=' | ')
    if visited[i]:
        return
    visited[i] = True

    for n in graph[i]:
        dfs(n)

dfs(0)

```

The corresponding output is:

```
Python >> 0 | 4 | 1 | 0 | 2 | 6 | 2 | 3 | 4 |
```

Breadth-First Search DFS

Definition:

Breadth-First Search (BFS) is a graph traversal algorithm that explores neighboring nodes first before moving to their children. It ensures the shortest path to each node in an unweighted graph. BFS uses a queue to maintain the order of node exploration, avoiding revisiting nodes and ensuring each node is visited only once.

Use cases:

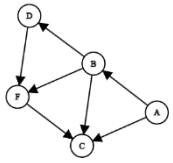
Breadth-First Search (BFS) is a versatile graph traversal algorithm widely used to find the shortest path, discover connections, and explore networks efficiently. It is crucial in route planning, social networking, web crawling, and puzzle-solving tasks, making it an essential tool for exploring and analyzing various structures and relationships in different fields.

Algorithm:

```
1. # Variables
2. graph = Adjacency List
3.
4. def bfs(graph, start):
5.     visited = set()
6.     queue = [start]
7.
8.     while queue:
9.         node = queue.pop(0)
10.        if node not in visited:
11.            print(node, end=' ')
12.            visited.add(node)
13.            queue.extend(graph[node])
14.
15. # Starting the BFS from node n
16. bfs(graph, n)
17.
```

Example:

Here's a small example illustrating an example of input outputs for the BFS:



We will use the Python code down below to outline the output of the algorithm on this graph:

```
# Variables
graph = {
    'A': ['B', 'C'],
    'B': ['C', 'D', 'F'],
    'D': ['F'],
    'F': ['C']
}

def bfs(graph, start):
    visited = set()
    queue = [start]

    while queue:
        node = queue.pop(0)
        if node not in visited:
            print(node, end=' ')
            visited.add(node)
            if node in graph.keys():
                queue.extend(graph[node])

# Starting the BFS from node 'A'
bfs(graph, 'A')

```

The corresponding output is:

```
Python >> A | B | C | D | F |
```

Topological Ordering Topsort

Definition:

Topological sorting (or Topsort) is a linear ordering of the vertices of a directed graph such that for every directed edge (u, v), vertex u comes before vertex v in the ordering. In other words, it arranges the vertices in such a way that all directed edges go from left to right.

Topological sorting is only possible for directed acyclic graphs (DAGs). If the graph contains cycles (i.e., there is a path from a vertex back to itself), then no valid topological sorting exists.

Use cases:

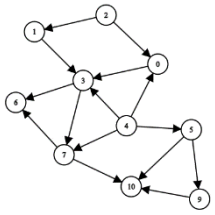
Topological sorting has various applications, including task scheduling, dependency resolution, and finding the order of execution in projects or workflows. It is a fundamental algorithm used in graph theory and has practical use in various fields such as project management and software engineering.

Algorithm:

```
 1. # Variables
 2. graph = adjacency list
15. 
16. # The Depth-First Search Algorithm
17. def dfs(arg, visited, graph):
18.     if arg in visited:
19.         return []
20.     else:
21.         visited.add(arg)
22.         l = []
23.         for i in graph[arg]:
24.             l = dfs(i, visited, graph) + l
25.         return [arg] + l
26. 
27. # The Actual Topological Ordering Algorithm
28. def topsort(graph):
29.     n = len(graph)
30.     visited = set()
31.     l = []
32.     for i in range(n):
33.         if not i in visited:
34.             l = dfs(i, visited, graph) + l
35.     return l
36. 
37. print(topsort(graph))
38. 
```

Example:

Here's a small example illustrating an example of input outputs for the BFS:



We will use the Python code down below to outline the output of the algorithm on this graph:

```
# Variables
graph = [
    [3],
    [3],
    [0, 1],
    [6, 7],
    [0, 3, 5, 7],
    [9, 10],
    [],
    [6, 10],
    [],
    [10],
    []
]

# The Depth-First Search Algorithm
def dfs(arg, visited, graph):
    if arg in visited:
        return []
    else:
        visited.add(arg)
        l = []
        for i in graph[arg]:
            l = dfs(i, visited, graph) + l
        return [arg] + l

# The Actual Topological Ordering Algorithm
def topsort(graph):
    n = len(graph)
    visited = set()
    l = []
    for i in range(n):
        if not i in visited:
            l = dfs(i, visited, graph) + l
    return l

print(topsort(graph))

```

The corresponding output is:

```
Python >> [8, 4, 5, 9, 2, 1, 0, 3, 7, 10, 6]
```

Single Source Shortest Path DAG SSSP

Definition:

Single Source Shortest Path (SSSP) is a fundamental problem in graph theory and network analysis, aiming to find the shortest path from a given source node to all other nodes in a weighted graph. The goal is to determine the minimum distance and the corresponding path between the source node and every other node in the graph.

The algorithm in this book works by first doing a topological ordering, assuming that all the paths are infinitely distant and initializing the first node distance to 0 then fixing the distance if we find a shorter one for every element. This algorithm works at a time complexity of $O(V + E)$.

Use cases:

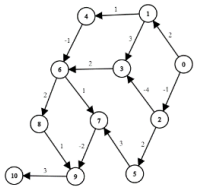
The SSSP DAG problem finds applications in various fields, such as route planning, network routing, and scheduling, where finding the shortest paths from a given starting point to all other reachable points is a fundamental optimization task.

Algorithm:

```
# Variables
graph = adjacency list

# SSSP DAG Algorithm
def SSSP(graph):
    n = len(graph)
    order = topsort(graph)
    distances = [float('inf')] * n
    distances[ order[0] ] = 0

    for arg in order:
        for el in graph[arg]:
            distances[el[0]] = min(distances[arg] + el[1], distances[el[0]])

    return order, distances

order, distances = SSSP(graph)
print(distances)
```

Example:

Here's a small example illustrating an example of input outputs for the SSSP DAG:



We will use the Python code down below to outline the output of the algorithm on this graph:

```
# Variables
graph = [
    [(1, 2), (2, -1)],
    [(3, 3), (4, 1)],
    [(3, 4), (5, 2)],
    [(6, 2)],
    [(6, 1)],
    [(7, 3)],
    [(7, 1), (8, 2)],
    [(9, -2)],
    [(9, 1)],
    [(10, 3)],
    []
]

# The Depth-First Search Algorithm
def dfs(arg, visited, graph):
    if arg in visited:
        return []
    else:
        visited.add(arg)
        l = []
        for el in graph[arg]:
            l = dfs(el[0], visited, graph) + l
        return [arg] + l

# The Topological Ordering Algorithm
def topsort(graph):
    n = len(graph)
    visited = set()
    l = []
    for i in range(n):
        if not i in visited:
            l = dfs(i, visited, graph) + l
    return l

# SSSP DAG Algorithm
def SSSP(graph):
    n = len(graph)
    order = topsort(graph)
    distances = [float('inf')] * n
    distances[ order[0] ] = 0

    for arg in order:
        for el in graph[arg]:
            distances[el[0]] = min(distances[arg] + el[1], distances[el[0]])

    return order, distances

order, distances = SSSP(graph)
print(order, distances, sep="\n")
```

The corresponding output is:

```
Python>> [0, 2, 5, 1, 4, 3, 6, 8, 7, 9, 10]
      >> [0, 2, -1, 3, 3, 1, 4, 4, 6, 2, 5]
```

Longest Path Algorithm:

Here you should just multiply all the edges by -1, find the shortest path using any SSSP algorithm and then remultiply them by -1 again and there you go, the longest path to each node

Bellman-Ford BF Algorithm

Definition:

The Bellman-Ford algorithm iteratively relaxes the edges, minimizing the distance to each vertex, and continues this process for V-1 iterations, where V is the number of vertices in the graph. During each iteration, the algorithm examines all edges and updates the distances to the vertices if a shorter path is found. This process is repeated until no further updates are possible, ensuring that the shortest distances have been calculated. Bellman-Ford guarantees the correct shortest paths. The time complexity of Bellman-Ford is $O(V\ E)$, where V is the number of vertices and E is the number of edges in the graph.

Use cases:

This Algorithm might not be the fastest SSSP algorithm but it comes in handy for determining negative cycles and where they occur.

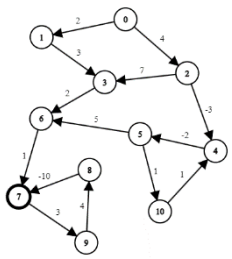
Detecting negative cycles is an essential task in various applications, including financial trading, where it helps identify opportunities for arbitrage. In network systems, finding negative cycles is crucial to prevent the occurrence of loops that could lead to instability or undesired feedback effects. Additionally, in graph algorithms, identifying negative cycles is fundamental for solving problems such as finding the shortest path with negative edge weights, where these cycles may affect the optimal route.

Algorithm:

```
 1. # Variables
 2. graph = adjacency list
 3.
 4. #Bellman-Ford Algorithm
 5. def bf(graph, start):
 6.     n = len(graph)
 7.     distances = [float('inf')] * n
 8.     distances[start] = 0
 9.     for i in range(n - 1):
10.         for arg in range(n):
11.             for to, weight in graph[arg]:
12.                 distances[ to ] = min(distances[ to ] , distances[arg] + weight)
13.     for i in range(n - 1):
14.         for arg in range(n):
15.             for to, weight in graph[arg]:
16.                 if distances[ to ] > distances[arg] + weight:
17.                     distances[ to ] = -float('inf')
18.
19.     return distances
20.
21. print( bf(graph, 0) )
22.
```

Example:

Here's a small example illustrating an example of input outputs for the Bellman-Ford:



We will use the Python code down below to outline the output of the algorithm on this graph:

```
# Variables
graph = [
    [(1, 2), (2, 4)],
    [(3, 3)],
    [(3, 7), (4, -3)],
    [(6, 2)],
    [(5, -2)],
    [(6, 5), (10, 1)],
    [(7, 1)],
    [(9, 3)],
    [(7, -10)],
    [(8, 4)],
    [(4, 1)]
]

#Bellman-Ford Algorithm
def bf(graph, start):
    n = len(graph)
    distances = [float('inf')] * n
    distances[start] = 0

    for i in range(n - 1):
        for arg in range(n):
            for to, weight in graph[arg]:
                distances[ to ] = min(distances[ to ] , distances[arg] + weight)

    for i in range(n - 1):
        for arg in range(n):
            for to, weight in graph[arg]:
                if distances[ to ] > distances[arg] + weight:
                    distances[ to ] = -float('inf')

    return distances

print( bf(graph, 0) )

```

The corresponding output is:

```
Python >> [0, 2, 4, 5, 1, -1, 4, -inf, -inf, -inf, 0]
```

Dijkstra's Algorithm

Definition:

Dijkstra's algorithm is a popular graph traversal and shortest path finding algorithm. Given a weighted graph with non-negative edge weights and a source node, it efficiently computes the shortest path from the source node to all other nodes in the graph. The algorithm maintains a priority queue of nodes, starting with the source node and iteratively expands to find the next closest node. It keeps track of the minimum distance from the source to each node and updates the distances whenever a shorter path is found. Dijkstra's algorithm is known for its optimality when dealing with non-negative edge weights and can be implemented using various data structures like priority queues or min-heaps to achieve time complexity of $O(E * \log V)$, where V is the number of vertices and E is the number of edges in the graph.

Lazy Dijkstra's Algorithm:

We first have to create a distances array where the distance to every node is positive infinity and the starting node has a distance of 0.

Then we should create a priority queue of key-value pairs of (node_index, distance) tuple which will help us figure out which node to visit based on sorted minimum value.

Then insert (s, 0) into the PQ and loop while PG is not empty pulling out the next most promising tuple.

Then Iterate over all edges outwards from the current node and relax them and appending the tuple to the PQ for every relaxation.

```
#Global Variables
graph = adjacency list
s = starting node index

#Dijkstra's Algorithm
def dijkstra(graph, start):
    n = len(graph)
    distances = [float('inf')] * n
    pq = PriorityQueue()
    visited = set()
    distances[start] = 0
    pq.put((start, 0))
    while not pq.empty():
        i, d = pq.get()
        visited.add(i)
        for el in graph[i]:
            if el[0] in visited:
                continue
            dist = d + el[1]
            if dist < distances[ el[0] ]:
                distances[ el[0] ] = dist
                pq.put( (el[0], dist) )
    return distances

print(dijkstra(graph, 0))
```

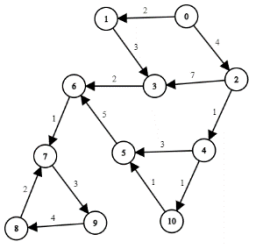
The reason we use an (index, value) tuple is because it's faster at a time complexity of $O(\log V)$ rather than using a key which is a lot slower at a linear time complexity.

Eager Dijkstra's Algorithm:

Lazy Dijkstra is called lazy because we lazily remove the edges that are out of date, therefore to optimize our task, we can use an indexed priority queue to update these elements. You can try that out on your own but the lazy Dijkstra is the one that is mostly used.

Example:

Here's a small example illustrating an example of input outputs for the Dijkstra:



We will use the Python code down below to outline the output of the algorithm on this graph:

```python
from queue import PriorityQueue

# Variables
graph = [
    [(1, 2), (2, 4)],
    [(3, 3)],
    [(3, 7), (4, 1)],
    [(6, 2)],
    [(5, 3), (10, 1)],
    [(6, 5)],
    [(7, 1)],
    [(9, 3)],
    [(7, 2)],
    [(8, 4)],
    [(5, 1)]
]

def dijkstra(graph, start):
    n = len(graph)
    distances = [float('inf')] * n
    pq = PriorityQueue()
    visited = set()

    distances[start] = 0
    pq.put((start, 0))

    while not pq.empty():
        i, d = pq.get()
        visited.add(i)

        for el in graph[i]:
            if el[0] in visited:
                continue
```

```
            dist = d + el[1]
            if dist < distances[ el[0] ]:
                distances[ el[0] ] = dist
                pq.put( (el[0], dist) )
    return distances

print(dijkstra(graph, 0))
```

The corresponding output is:

```
Python>> [0, 2, 4, 5, 5, 7, 7, 8, 15, 11, 6]
```

Floyd-Warshall FW Algorithm

Definition:

The Floyd-Warshall algorithm is a dynamic programming approach used to find the shortest paths between all pairs of vertices in a weighted graph. It works on both directed and undirected graphs and efficiently handles negative edge weights, making it suitable for graphs with negative cycles. By iteratively considering all intermediate vertices as potential shortcuts between two vertices, the algorithm efficiently computes the shortest distances between all pairs of vertices, resulting in a matrix containing the shortest paths.

Use cases:

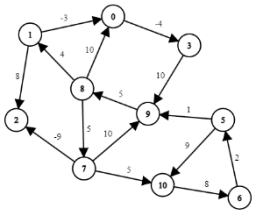
The Floyd-Warshall algorithm efficiently computes the shortest paths between all pairs of vertices in a single pass, distinguishing it from other algorithms that focus on single-source or single-destination paths. By employing dynamic programming and considering intermediate vertices as potential shortcuts, it creates a matrix with shortest distances, enabling quick retrieval of shortest paths for any pair of vertices in a weighted graph. This makes it a powerful choice for scenarios requiring all-pairs shortest path calculations.

Algorithm:

```
 1. # Variables
 2. inf = float('inf')
 3.
 4. #APSP Floyd-Warshall Algorithm
 5. def fw(graph):
 6.     n = len(graph)
 7.     r = [[float('inf') if i != j else 0 for j in range(n)] for i in range(n)]
 8.
 9.     for i in range(n):
10.         for j in range(n):
11.             if not graph[i][j] == 0:
12.                 r[i][j] = graph[i][j]
13.     # The Actual Algorithm
14.     for k in range(n):
15.         for i in range(n):
16.             for j in range(n):
17.                 r[i][j] = min(r[i][k] + r[k][j], r[i][j])
18.
19.     # Detect negative cycles
20.     for k in range(n):
21.         for i in range(n):
22.             for j in range(n):
23.                 if r[i][j] > r[i][k] + r[k][j]:
24.                     r[i][j] = -inf
25.     return r
26.
27. show(fw(graph))
```

Example:

Here's a small example illustrating an example of input outputs for the Floyd-Warshall FW:



We will use the Python code down below to outline the output of the algorithm on this graph:

```
# Variables
inf = float('inf')
graph = [
    #  0    1    2    3    4    5    6    7    8    9    10
    [  0,   0,   0,  -4,   0,   0,   0,   0,   0,   0,   0], # 0
    [ -3,   0,   8,   0,   0,   0,   0,   0,   0,   0,   0], # 1
    [  0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0], # 2
    [  0,   0,   0,   0,   0,   0,   0,   0,   0,  10,   0], # 3
    [  0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0], # 4
    [  0,   0,   0,   0,   0,   0,   0,   0,   0,   1,   9], # 5
    [  0,   0,   0,   0,   0,   2,   0,   0,   0,   0,   0], # 6
    [  0,   0,  -9,   0,   0,   0,   0,   0,   0,  10,   5], # 7
    [ 10,   4,   0,   0,   0,   0,   0,   5,   0,   0,   0], # 8
    [  0,   0,   0,   0,   0,   0,   0,   0,   5,   0,   0], # 9
    [  0,   0,   0,   0,   0,   0,   8,   0,   0,   0,   0], # 10
]

#APSP Floyd-Warshall Algorithm
def fw(graph):
    n = len(graph)
    r = [[float('inf') if i != j else 0 for j in range(n)] for i in range(n)]

    for i in range(n):
        for j in range(n):
            if not graph[i][j] == 0:
                r[i][j] = graph[i][j]
    # The Actual Algorithm
    for k in range(n):
        for i in range(n):
            for j in range(n):
                r[i][j] = min(r[i][k] + r[k][j], r[i][j])

    # Detect negative cycles
    for k in range(n):
        for i in range(n):
            for j in range(n):
                if r[i][j] > r[i][k] + r[k][j]:
                    r[i][j] = -inf
    return r

show(fw(graph))

```

The corresponding output is:

```
Python>> 00 15 07 -4 +∞ 31 29 16 11 06 21
      >> -3 00 04 -7 +∞ 28 26 13 08 03 18
      >> +∞ +∞ 00 +∞ +∞ +∞ +∞ +∞ +∞ +∞ +∞
      >> 16 19 11 00 +∞ 35 33 20 15 10 25
      >> +∞ +∞ +∞ +∞ 00 +∞ +∞ +∞ +∞ +∞ +∞
      >> 07 10 02 03 +∞ 00 17 11 06 01 09
      >> 09 12 04 05 +∞ 02 00 13 08 03 11
      >> 16 19 -9 12 +∞ 15 13 00 15 10 05
      >> 01 04 -4 -3 +∞ 20 18 05 00 07 10
      >> 06 09 01 02 +∞ 25 23 10 05 00 15
      >> 17 20 12 13 +∞ 10 08 21 16 11 00
```

Bridges and Articulation Points

Definition:

Bridges, also known as cut edges, are edges in an undirected graph that, if removed, increase the number of connected components in the graph. In other words, a bridge is an edge whose removal would disconnect the graph or split it into two or more separate components. Bridges are critical for maintaining connectivity in the graph, and their identification is essential for understanding the structure and robustness of the graph.

Articulation points, also known as cut vertices, are vertices in an undirected graph that, if removed along with their incident edges, increase the number of connected components in the graph. Similar to bridges, an articulation point is a vertex whose removal would disconnect the graph or split it into multiple components. Articulation points play a crucial role in determining the connectivity of the graph and identifying important nodes that act as "hinges" between different parts of the graph.

Use cases:

Bridges and articulation points have significant applications in graph analysis and network-related scenarios. Bridges are critical edges in a graph whose removal disconnects the graph, making them essential in network design, social network analysis, and transportation planning. On the other hand, articulation points are vertices whose removal disconnects the graph into multiple components. They are used to assess network reliability, optimize internet routing, and plan emergency responses. Identifying and understanding bridges and articulation points help in improving network performance and designing resilient and efficient systems.

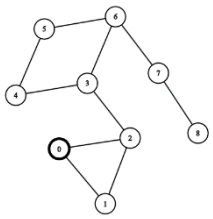
Algorithm For Finding Bridges:

```
#Variables
graph = adjacency list
#Global variables needed for the Bridges Search Algorithm
id = 0

# DFS Algorithm
def dfs(graph, i, p, visited, low_links, ids, bridges):
    global id
    id+=1
    visited[i] = True
    low_links[i] = ids[i] = id
    for arg in graph[i]:
        if arg == p: continue
        if not visited[arg]:
            dfs(graph, arg, i, visited, low_links, ids, bridges)
            low_links[i] = min(low_links[i], low_links[arg])
            if ids[i] < low_links[arg]:
                bridges.append((i, arg))
        else:
            low_links[i] = min(low_links[i], low_links[arg])


```

```
# Bridges Search Algorithm
def bridges_search(graph):
    n = len(graph)
    visited = [False] * n
    low_links = [0] * n
    ids = [0] * n
    bridges = []
    for i in range(n):
        if not visited[i]: dfs(graph, i, -1, visited, low_links, ids, bridges)
    return bridges

print(bridges_search(graph))

```

Example:

Here's a small example illustrating an example of input outputs for the Bridges Finding Algorithm:



We will use the Python code down below to outline the output of the algorithm on this graph:

```
#Variables
graph =[
    [1, 2],
    [0, 2],
    [0, 1, 3],
    [2, 4, 6],
    [3, 5],
    [4, 6],
    [3, 5, 7],
    [6, 8],
    [7]
]
#Global variables needed for the Bridges Search Algorithm
id = 0
# DFS Algorithm
def dfs(graph, i, p, visited, low_links, ids, bridges):
    global id
    id+=1
    visited[i] = True
    low_links[i] = ids[i] = id
    for arg in graph[i]:
        if arg == p: continue
        if not visited[arg]:
            dfs(graph, arg, i, visited, low_links, ids, bridges)
            low_links[i] = min(low_links[i], low_links[arg])
            if ids[i] < low_links[arg]:
                bridges.append((i, arg))
        else:
            low_links[i] = min(low_links[i], low_links[arg])
```

```
# Bridges Search Algorithm
def bridges_search(graph):
    n = len(graph)
    visited = [False] * n
    low_links = [0] * n
    ids = [0] * n
    bridges = []
    for i in range(n):
        if not visited[i]: dfs(graph, i, -1, visited, low_links, ids, bridges)
    return bridges

print(bridges_search(graph))
```

The corresponding output is:

```
Python >> [(7, 8), (6, 7), (2, 3)]
```

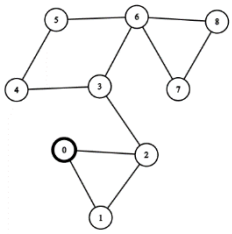
Algorithm For Finding Articulation Points:

The process of finding an articulation point is similar to the one of finding bridges however there are exceptions where the articulation point is trapped into a cycle or it's a singleton so we will have to put some changes in the algorithm.

```
#Variables
graph =adjacency list
#Global variables needed for the Articulation Points Search Algorithm
id = 0
out = 0
# DFS Algorithm
def dfs(graph, r, i, p, visited, low_links, ids, art):
    global id, out
    visited[i] = True
    low_links[i] = ids[i] = id
    id += 1
    for arg in graph[i]:
        if arg == p: continue
        if not visited[arg]:
            dfs(graph, r, arg, i, visited, low_links, ids, art)
            low_links[i] = min(low_links[i], low_links[arg])
            if ids[i] <= low_links[arg]:
                art[i] = True
        else:
            low_links[i] = min(low_links[i], low_links[arg])
# Articulation Points Search Algorithm
def art_search(graph):
    global out
    n = len(graph)
    visited = [False] * n
    low_links = [0] * n
    ids = [0] * n
    art = [False] * n
    for i in range(n):
        if not visited[i]:
            out = 0
            dfs(graph, i, i, -1, visited, low_links, ids, art)
            art[i] = out > 1
    return art

print(art_search(graph))

```

Example:

Here's a small example illustrating an example of input outputs for the Articulation points Finding Algorithm:



We will use the Python code down below to outline the output of the algorithm on this graph:

```
#Variables
graph =[
    [1, 2],
    [0, 2],
    [0, 1, 3],
    [2, 4, 6],
    [3, 5],
    [4, 6],
    [3, 5, 7, 8],
    [6, 8],
    [6, 7]
]
#Global variables needed for the Articulation Points Search Algorithm
id = 0
out = 0
# DFS Algorithm
def dfs(graph, r, i, p, visited, low_links, ids, art):
    global id, out
    visited[i] = True
    low_links[i] = ids[i] = id
    id += 1
    for arg in graph[i]:
        if arg == p: continue
        if not visited[arg]:
            dfs(graph, r, arg, i, visited, low_links, ids, art)
            low_links[i] = min(low_links[i], low_links[arg])
            if ids[i] <= low_links[arg]:
                art[i] = True
        else:
            low_links[i] = min(low_links[i], low_links[arg])
# Articulation Points Search Algorithm
def art_search(graph):
    global out
    n = len(graph)
    visited = [False] * n
    low_links = [0] * n
    ids = [0] * n
    art = [False] * n
    for i in range(n):
        if not visited[i]:
            out = 0
            dfs(graph, i, i, -1, visited, low_links, ids, art)
            art[i] = out > 1
    return art
print(art_search(graph))
```

The corresponding output is:

```
Python >> [True, False, True, True, False, False, False, False, False]
```

Tarjan’s Algorithm

Definition:

Tarjan's algorithm is a popular graph traversal algorithm used for finding strongly connected components in directed graphs. Developed by Robert Tarjan in 1972, this algorithm efficiently determines sets of vertices that have mutual paths between them.

The core idea behind Tarjan's algorithm is to perform a depth-first search (DFS) on the graph while maintaining information about the depth of each vertex and the earliest reachable ancestor of each vertex. Through this process, the algorithm can identify and group together vertices that form strongly connected components. Strongly connected components are essential for various graph-related problems, such as analyzing network structures, identifying cycles, and solving certain optimization tasks. Tarjan's algorithm achieves a linear runtime complexity of $O(V + E)$, making it a powerful tool for graph analysis.

Use cases:

Tarjan's algorithm finds extensive applications in the domain of computer science and graph theory due to its ability to identify strongly connected components efficiently. One of its primary use cases is in identifying cycles within directed graphs, which is crucial for detecting potential deadlocks and analyzing iterative processes in various systems. Additionally, Tarjan's algorithm is employed in network analysis to identify groups of closely connected nodes or communities, facilitating efficient information flow and network optimization. This algorithm also plays a vital role in identifying critical points, such as articulation points and bridges, in undirected graphs. By pinpointing these critical points, it becomes possible to evaluate the robustness and resilience of complex systems. In summary, Tarjan's algorithm is a versatile and powerful tool for analyzing graphs and networks, enabling researchers and engineers to gain valuable insights into the underlying structures and connections present in diverse datasets and systems.

Algorithm:

Begin by marking each node's ID as unvisited.

Initiate a Depth-First Search (DFS) traversal. During traversal, assign an ID and a low-link value to each visited node. Additionally, keep track of visited nodes by adding them to a seen stack.

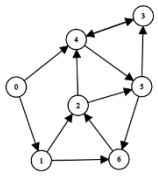
During the callback phase of DFS, check if the previous node exists in the stack. If it does, update the current node's low-link value to the minimum between its current value and the low-link value of the previous node.

After exploring all neighbors of a node, check if the current node initiated a connected component. If it did, remove nodes from the stack until the current node is reached, effectively identifying and extracting the strongly connected component.

```
#Variables
graph = Adjacency List
#Global Variables and constants
UNVISITED = -1

class Tarjan:
    def findSccs(self, graph):
        self.n = len(graph)
        self.sccs =0
        self.id = 0
        self.ids = [UNVISITED] * self.n
        self.low_links = [0] * self.n
        self.stacked = [False] * self.n
        self.stack = []

        for i in range(self.n):
            if self.ids[i] == UNVISITED:
                self.dfs(i, graph)
        return self.low_links

    def dfs(self, i, graph):
        self.stack.append(i)
        self.stacked[i] = True
        self.ids[i] = self.low_links[i] = self.id
        self.id += 1
        for arg in graph[i]:
            if self.ids[i] == UNVISITED: self.dfs(arg)
            if self.stacked[arg]: self.low_links[i] = min(self.low_links[arg],
                                                          self.low_links[i])
        if self.ids[i] == self.low_links[i]:
            node = self.stack.pop()
            while not node == i:
                self.stacked[node] = False
                self.low_links[node] = self.ids[i]
                node = self.stack.pop()
            self.sccs += 1

print(Tarjan().findSccs(graph))
```

Example:

Here's a small example illustrating an example of input outputs for the Tarjan's Algorithm:



We will use the Python code down below to outline the output of the algorithm on this graph:

```
#Variables
graph =[
    [1, 4],
    [2, 6],
    [4, 5],
    [4],
    [5, 3],
    [3, 6],
    [2]
]
#Global Variables and constants
UNVISITED = -1

class Tarjan:
    def findSccs(self, graph):
        self.n = len(graph)
        self.sccs =0
        self.id = 0
        self.ids = [UNVISITED] * self.n
        self.low_links = [0] * self.n
        self.stacked = [False] * self.n
        self.stack = []

        for i in range(self.n):
            if self.ids[i] == UNVISITED:
                self.dfs(i, graph)
        return self.low_links

    def dfs(self, i, graph):
        self.stack.append(i)
        self.stacked[i] = True
        self.ids[i] = self.low_links[i] = self.id
        self.id += 1
        for arg in graph[i]:
            if self.ids[i] == UNVISITED: self.dfs(arg)
            if self.stacked[arg]: self.low_links[i] = min(self.low_links[arg],
                                                          self.low_links[i])
        if self.ids[i] == self.low_links[i]:
            node = self.stack.pop()
            while not node == i:
                self.stacked[node] = False
                self.low_links[node] = self.ids[i]
                node = self.stack.pop()
            self.sccs += 1

print(Tarjan().findSccs(graph))
```

The corresponding output is:

```
Python >> [0, 1, 2, 3, 3, 3, 2]
```

Eulerian Paths

Definition:

Eulerian paths are trails in a graph that visit each edge exactly once but may not necessarily return to the starting vertex. To find an Eulerian path in a graph, we look for a vertex with an odd degree (the number of edges incident to it). If there are exactly two vertices with an odd degree, one of them will be the starting vertex, and the other will be the ending vertex of the Eulerian path. If there are no vertices with an odd degree, the graph may have an Eulerian circuit, which is a closed path that visits all edges and vertices exactly once.

Eulerian circuits, also known as Eulerian cycles, are a special case of Eulerian paths where the path starts and ends at the same vertex. This means that an Eulerian circuit traverses all edges and vertices of the graph exactly once and forms a closed loop. A graph has an Eulerian circuit if and only if every vertex has an even degree..

Use cases:

The applications of Eulerian paths and circuits are vast and varied. In transportation planning, these concepts help optimize routes for vehicles and goods distribution, ensuring efficient logistics and reduced costs. In computer networks and communication systems, they play a vital role in designing algorithms for data routing and transmission, improving network performance and reducing congestion. In biology, Eulerian paths and circuits are used in DNA sequencing and genome assembly, aiding in understanding genetic information and evolutionary relationships. Additionally, they find applications in electrical engineering for circuit analysis and design, as well as in social network analysis and various other fields where interconnected structures need to be efficiently navigated.

Conditions of Existence:

To find Eulerian paths and circuits, there are rules that the graph has to conform with. Here's a table that explains everything:

	Eulerian Circuit	**Eulerian Path**
Undirected Graph	Every node has an even degree.	Either it's an Eulerian circuit or there are exactly two nodes that have odd degree
Directed Graph	Every node has equal indegree and outdegree	At most one vertex has (outdegree) – (indegree)=1 and at most one vertex has (indegree) – (outdegree) = 1 and all other vertices have equal in and out degrees

A degree of a node is the number of edges it has, a indegree is the number of input edges it has and the outdegree is the number output edges it has. Note that the graph should be a single component.

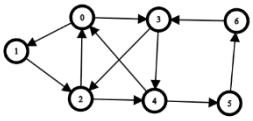
Algorithm For Finding an Eulerian Path and/or Circuit:

Before finding the Eulerian path in a graph, we need to make sure it exists first, then we should find the right starting and ending nodes, the starting node is the node that has an extra outgoing edge and the ending node is the one that has one extra ingoing edge. If all nodes have equal in and out degrees then we have an Eulerian circuit in which we can start at any random node. In this algorithm we're going to use an edited version of DFS algorithm to find the path. This algorithm runs at a time complexity of $O(E)$.

```
#Variables and constants
graph = adjacency list
def degrees(graph, n):
    indegrees = [0] * n
    outdegrees = [0] * n
    for i in range(n):
        outdegrees[i] = len(graph[i])
        for el in graph[i]:
            indegrees[el] += 1
    return indegrees, outdegrees

def circuitexists(indegrees, outdegrees):
    for i,o in zip(indegrees, outdegrees):
        if not i==o: return False
    return True

def pathexists(indegrees, outdegrees):
    count1 = count2 = 0
    for i,o in zip(indegrees, outdegrees):
        if count1>1 or count2>1 or abs(i-o)>1: return False
        count1 += i-o == 1
        count2 += o-i == 1
    return True

def findstart(indegrees, outdegrees, n):
    start = 0
    for i in range(n):
        if outdegrees[i] - indegrees[i] == 1: return i
        if outdegrees[i]>0: start = i
    return start

def dfs(graph, i, path, outdegrees):
    while not outdegrees[i] == 0:
        outdegrees[i] -= 1
        dfs(graph, graph[i][outdegrees[i]] , path, outdegrees)
    path.insert(0, i)

def findpath(graph):
    path = []
    n = len(graph)
    indegrees, outdegrees = degrees(graph, n)
    if pathexists(indegrees, outdegrees):
        dfs(graph, findstart(indegrees, outdegrees, n), path, outdegrees)
    return path

print(findpath(graph))
```

Example:

Here's a small example illustrating an example of input outputs for the Eulerian Circuit Finding Algorithm:



We will use the Python code down below to outline the output of the algorithm on this graph:

```python
#Variables and constants
graph = [
    [1, 3],
    [2],
    [0, 4],
    [2, 4],
    [0, 5],
    [6],
    [3]
]
def degrees(graph, n):
    indegrees = [0] * n
    outdegrees = [0] * n
    for i in range(n):
        outdegrees[i] = len(graph[i])
        for el in graph[i]:
            indegrees[el] += 1
    return indegrees, outdegrees
def circuitexists(indegrees, outdegrees):
    for i,o in zip(indegrees, outdegrees):
        if not i==o: return False
    return True
def pathexists(indegrees, outdegrees):
    count1 = count2 = 0
    for i,o in zip(indegrees, outdegrees):
        if count1>1 or count2>1 or abs(i-o)>1: return False
        count1 += i-o == 1
        count2 += o-i == 1
    return True
def findstart(indegrees, outdegrees, n):
    start = 0
    for i in range(n):
        if outdegrees[i] - indegrees[i] == 1: return i
        if outdegrees[i]>0: start = i
    return start
def dfs(graph, i, path, outdegrees):
    while not outdegrees[i] == 0:
        outdegrees[i] -= 1
        dfs(graph, graph[i][outdegrees[i]] , path, outdegrees)
    path.insert(0, i)
def findpath(graph):
    path = []
    n = len(graph)
    indegrees, outdegrees = degrees(graph, n)
    if pathexists(indegrees, outdegrees):
        dfs(graph, findstart(indegrees, outdegrees, n), path, outdegrees)
    return path
print(findpath(graph))
```

The corresponding output is:

```
1. Python >> [6, 3, 4, 0, 3, 2, 0, 1, 2, 4, 5, 6]
```

Traveling Salesman Problem TSP with DP

Definition:

The Traveling Salesman Problem (TSP) is a well-known optimization problem in graph theory and combinatorial optimization. The goal of the TSP is to find the shortest possible route that visits a given set of cities exactly once and returns to the starting city. The problem is NP-hard, which means that there is no known polynomial-time algorithm to solve it for large instances. However, there are several heuristic and approximation algorithms that can find reasonably good solutions in a reasonable amount of time for practical instances.

One of the most popular algorithms for solving the TSP is the Held-Karp algorithm, also known as the dynamic programming algorithm which is what we're going to implement in this book. The Held-Karp algorithm works by breaking the problem into smaller subproblems and solving them recursively. It maintains a table to store the optimal solutions to these subproblems and uses them to build the final solution for the original problem. The time complexity of the Held-Karp algorithm is $O(n^2 2^n)$, where n is the number of cities in the TSP instance. While this is an improvement over the naive brute-force approach (which has a time complexity of O(n!), the Held-Karp algorithm is still not efficient enough for large instances.

Use cases:

The Traveling Salesman Problem has numerous real-world applications in various fields, including logistics, transportation, and network design. For example, in logistics, it can be used to optimize delivery routes for a fleet of vehicles to minimize travel time and costs. In transportation, it can help plan efficient routes for public transport systems. In network design, it can be used to find the shortest path for data transmission in communication networks. The TSP also has applications in manufacturing, where it can be used to optimize the order in which different tasks are performed to minimize production time. Despite its computational complexity, the TSP's wide range of applications makes it an essential problem to study and develop efficient approximation algorithms to find near-optimal solutions in practical scenarios.

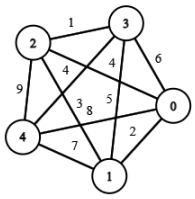
Algorithm:

```
#Variables and constants
adj_matrix = adjacency matrix
INF = float('inf')
# This recursive method is used in the next function to create bit sets
def combinations(s, i, r, n, subs):
    if n - i < r: return
    if r == 0:
        subs.append(s)
    else:
        for j in range(i, n):
            s = s | 1<<j
            combinations(s, i+1, r-1, n, subs)
            s = s & ~(1<<j)
# returns all combinations of size N where there are r bits set to 1
```

```
# subsets(3, 4) = [0111, 1011, 1101, 1110]
def subsets(r, n):
    subs = []
    combinations(0, 0, r, n, subs)
    return subs
# ith bit in subset == 0 ?
def izero(i, subset):
    return ((1<<i) & subset) == 0
def tsp(graph, S):
    n = len(graph)
    # initialize table with null or -1 or +inf to prevent errors
    dp = [ [None] * (1<<n) for _ in range(n) ]
    # Catch the optimal solution from start node to others
    for i in range(n):
        if i == S: continue
        # Store the optimal value from node S to each node i
        dp[i][ 1<<S | 1<<i ] = graph[S][i]
    # solve
    for r in range(3, n+1):
        for subset in subsets(r, n):
            if izero(S, subset): continue
            for next in range(n):
                if next==S or izero(next, subset): continue
                #subset state without next node
                state = subset ^ (1<<next)
                minDist = INF
                for e in range(n):
                    if e == S or e == next or izero(e, subset): continue
                    minDist = min(dp[e][state] + graph[e][next], minDist)
                dp[next][subset] = minDist
    # Find minimum cost
    # the end state is the bit mask with n bits set to 1
    END_STATE = (1<<n)-1
    mc = INF
    for e in range(n):
        if e==S: continue
        mc = min(dp[e][END_STATE] + graph[e][S], mc)
    # Now Just find the optimal tour
    lastindex = S
    state = END_STATE
    tour = [S]
    for i in range(1, n):
        index = -1
        for j in range(n):
            if j==S or izero(j, state): continue
            if index==-1: index = j
            prev = dp[index][state] + graph[index][lastindex]
            new = dp[j][state] + graph[j][lastindex]
            if new < prev: index = j
        tour.append(index)
        state = state ^ (1<<index)
        lastindex = index
    tour.append(S)
    tour.reverse()
    return (mc, tour)

print(tsp(adj_matrix, 0))
```

Example:

Here's a small example illustrating an example of input outputs for the Held-Karp Algorithm:



We will use the Python code down below to outline the output of the algorithm on this graph:

```
#Variables and constants
adj_matrix = [
    [0, 2, 4, 6, 8],
    [2, 0, 3, 5, 7],
    [4, 3, 0, 1, 9],
    [6, 5, 1, 0, 4],
    [8, 7, 9, 4, 0]
]
INF = float('inf')
# This recursive method is used in the next function to create bit sets
def combinations(s, i, r, n, subs):
    if n - i < r: return
    if r == 0:
        subs.append(s)
    else:
        for j in range(i, n):
            s = s | 1<<j
            combinations(s, i+1, r-1, n, subs)
            s = s & ~(1<<j)
# returns all combinations of size N where there are r bits set to 1
# subsets(3, 4) = [0111, 1011, 1101, 1110]
def subsets(r, n):
    subs = []
    combinations(0, 0, r, n, subs)
    return subs
# ith bit in subset == 0 ?
def izero(i, subset):
    return ((1<<i) & subset) == 0

def tsp(graph, S):
    n = len(graph)
    # initialize table with null or -1 or +inf to prevent errors
    dp = [ [None] * (1<<n) for _ in range(n) ]
    # Catch the optimal solution from start node to others
    for i in range(n):
        if i == S: continue
        # Store the optimal value from node S to each node i
        dp[i][ 1<<S | 1<<i ] = graph[S][i]
```

```
    # solve
    for r in range(3, n+1):
        for subset in subsets(r, n):
            if izero(S, subset): continue
            for next in range(n):
                if next==S or izero(next, subset): continue
                #subset state without next node
                state = subset ^ (1<<next)
                minDist = INF
                for e in range(n):
                    if e == S or e == next or izero(e, subset): continue
                    minDist = min(dp[e][state] + graph[e][next], minDist)
                dp[next][subset] = minDist
    # Find minimum cost
    # the end state is the bit mask with n bits set to 1
    END_STATE = (1<<n)-1
    mc = INF
    for e in range(n):
        if e==S: continue
        mc = min(dp[e][END_STATE] + graph[e][S], mc)
    # Now Just find the optimal tour
    lastindex = S
    state = END_STATE
    tour = [S]
    for i in range(1, n):
        index = -1
        for j in range(n):
            if j==S or izero(j, state): continue
            if index==-1: index = j
            prev = dp[index][state] + graph[index][lastindex]
            new = dp[j][state] + graph[j][lastindex]
            if new < prev: index = j
        tour.append(index)
        state = state ^ (1<<index)
        lastindex = index
    tour.append(S)
    tour.reverse()
    return (mc, tour)
print(tsp(adj_matrix, 0))
```

The corresponding output is:

```
Python >> (18, [0, 1, 4, 3, 2, 0])
```

Ford Fulkerson Network Flow

Definition:

Ford-Fulkerson is a widely used algorithm for solving the maximum flow problem in network flow theory. The network flow problem involves finding the maximum amount of flow that can be sent from a source node to a sink node through a directed graph with capacities assigned to its edges. The main idea behind the Ford-Fulkerson algorithm is to find augmenting paths in the graph, which are paths from the source to the sink that can increase the flow. In each iteration, the algorithm finds an augmenting path using techniques like depth-first search or breadth-first search and computes the maximum amount of flow that can be pushed along this path. The process continues until no more augmenting paths can be found, and the maximum flow is achieved. The algorithm can be further improved using various techniques like the Edmonds-Karp algorithm, which guarantees a polynomial time complexity by using a shortest path search.

Use cases:

In network flow theory, a flow graph represents a network where the edges have capacities that indicate the maximum amount of flow they can carry. The goal is to find the maximum flow from a source node to a sink node while respecting the capacity constraints on each edge. The Ford-Fulkerson algorithm is an essential method to solve this problem. An augmenting path is a simple path from the source to the sink in the flow graph, and finding such paths is crucial in increasing the flow. The algorithm increases the flow along these paths until no more augmenting paths can be found, which ensures an optimal solution for the maximum flow problem. Network flow problems and the Ford-Fulkerson algorithm have numerous applications in real-world scenarios, such as traffic flow optimization, network bandwidth allocation, resource management in supply chains, and finding the maximum flow of data in computer networks. Additionally, the concept of flow graphs is used in various optimization problems like matching algorithms, image segmentation, and even in some mathematical proofs.

Algorithm:

1. Initialize the flow of all edges to zero.

2. While there exists an augmenting path from the source to the sink:

    a. Find an augmenting path using a graph traversal algorithm like DFS or BFS.

    b. Determine the minimum capacity (residual capacity) along this path. This is the maximum additional flow that can be pushed through the path.

    c. Update the flow of each edge along the augmenting path by adding the minimum capacity.

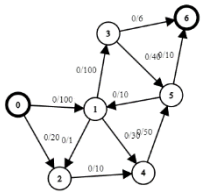
    d. Update the residual capacities of forward and backward edges. The residual capacity of a forward edge is the original capacity minus the flow, and for a backward edge, it is the flow.

3. When no more augmenting paths can be found, the algorithm terminates, and the current flow represents the maximum flow from the source to the sink.

```
# Constants
INF = float('inf')
# Edge Class
class Edge:
    def __init__(self, back, front, capactiy):
        self.back = back
        self.front = front
        self.capacity = capactiy
        self.residual = None
        self.flow = 0
    def isResidual(self):
        return self.capacity == 0
    def remaining_capactiy(self):
        return self.capacity - self.flow
    def augment(self, bottleNeck):
        self.flow += bottleNeck
        self.residual.flow -= bottleNeck

class FlowNetwork:
    def __init__(self, n, source, sink):
        self.n = n
        self.source = source
        self.sink = sink
        self.graph = [[] for _ in range(n)]
        self.visited = [0] * n
        self.visitedToken = 1
        self.max_flow = 0

    def add_edge(self, back, front, capacity):
        edge = Edge(back, front, capacity)
        residual = Edge(front, back, 0)
        edge.residual = residual
        residual.residual = edge
        self.graph[back].append(edge)
        self.graph[front].append(residual)
    # Ford Fulkerson Algorithm
    def dfs(self, node, flow):
        if node == self.sink: return flow

        self.visited[node] = self.visitedToken
        edges = self.graph[node]
        for edge in edges:
            if edge.remaining_capactiy()>0 and self.visited[edge.front] != self.visitedToken:
                bottlneck = self.dfs(edge.front, min(flow, edge.remaining_capactiy()))
                if bottlneck >0:
                    edge.augment(bottlneck)
                    return bottlneck
        return 0

    def find_max_flow(self):
        f = self.dfs(self.source, INF)
        while f!=0:
            self.visitedToken += 1
            self.max_flow += f
            f = self.dfs(self.source, INF)
        return self.max_flow
```

Example:

Here's a small example illustrating an example of input outputs for the Ford Fulkerson Algorithm:



We will use the Python code down below to outline the output of the algorithm on this graph:

```python
# Variables
INF = float('inf')
# Edge Class
class Edge:
    def __init__(self, back, front, capactiy):
        self.back = back
        self.front = front
        self.capacity = capactiy
        self.residual = None
        self.flow = 0
    def isResidual(self):
        return self.capacity == 0
    def remaining_capactiy(self):
        return self.capacity - self.flow
    def augment(self, bottleNeck):
        self.flow += bottleNeck
        self.residual.flow -= bottleNeck

class FlowNetwork:
    def __init__(self, n, source, sink):
        self.n = n
        self.source = source
        self.sink = sink
        self.graph = [[] for _ in range(n)]
        self.visited = [0] * n
        self.visitedToken = 1
        self.max_flow = 0

    def add_edge(self, back, front, capacity):
        edge = Edge(back, front, capacity)
        residual = Edge(front, back, 0)
        edge.residual = residual
        residual.residual = edge
        self.graph[back].append(edge)
        self.graph[front].append(residual)
    # Ford Fulkerson Algorithm
    def dfs(self, node, flow):
        if node == self.sink: return flow

        self.visited[node] = self.visitedToken
        edges = self.graph[node]
        for edge in edges:
            if edge.remaining_capactiy()>0 and self.visited[edge.front] != self.visitedToken:
                bottlneck = self.dfs(edge.front, min(flow, edge.remaining_capactiy()))
                if bottlneck >0:
                    edge.augment(bottlneck)
                    return bottlneck
        return 0

```

```
    def find_max_flow(self):
        f = self.dfs(self.source, INF)
        while f!=0:
            self.visitedToken += 1
            self.max_flow += f
            f = self.dfs(self.source, INF)
        return self.max_flow

#Application
ford = FlowNetwork(7, 0, 6)
ford.add_edge(0, 1, 100)
ford.add_edge(0, 2, 20)
ford.add_edge(1, 2, 1)
ford.add_edge(1, 3, 100)
ford.add_edge(1, 4, 30)
ford.add_edge(2, 4, 10)
ford.add_edge(3, 5, 40)
ford.add_edge(3, 6, 6)
ford.add_edge(4, 5, 50)
ford.add_edge(5, 1, 10)
ford.add_edge(5, 6, 10)

print(ford.find_max_flow())
```

The corresponding output is:

```
Python>> 16
```

Maximum Cardinality of Bipartite Matching MCBM

Definition:

The maximum cardinality bipartite matching problem is a combinatorial optimization problem that aims to find the maximum number of pairwise non-adjacent edges (matches) in a bipartite graph. In the context of matching, one set represents a group of agents or elements, and the other set represents another group of agents or elements. The goal is to find the largest possible set of pairs where each pair consists of one element from each group and no two pairs share a common element.

To solve the maximum cardinality bipartite matching problem, we can use the Ford-Fulkerson algorithm with a few modifications. First, we create a flow graph from the given bipartite graph, where we add a source node connected to all elements in the first set and a sink node connected to all elements in the second set. The capacity of each edge is set to 1, indicating that each edge can only carry one unit of flow. Next, we apply the Ford-Fulkerson algorithm to find the maximum flow from the source to the sink in this flow graph. The maximum flow value corresponds to the maximum number of pairwise non-adjacent edges (matches) in the original bipartite graph, which represents the solution to the maximum cardinality bipartite matching problem. The Ford-Fulkerson algorithm efficiently computes the maximum flow in the flow graph by iteratively finding augmenting paths and increasing the flow along these paths until no more augmenting paths can be found, ensuring that we obtain the optimal solution to the maximum cardinality bipartite matching problem.

Use cases:

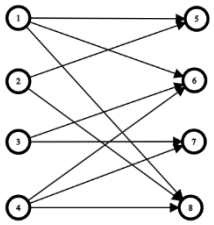
The maximum cardinality bipartite matching problem has diverse applications in different fields. It is used to optimize resource allocation in computer science and information technology, transportation and logistics, and bioinformatics. In job assignment scenarios, it helps match job seekers with vacancies, while in transportation, it optimizes cargo and vehicle assignments. In bioinformatics, it analyzes molecular interactions, and in social network analysis, it connects individuals in social networks. The problem's versatility makes it a valuable tool for solving various optimization problems in different domains.

Example Algorithm:

In this algorithm we will only use the Ford-Fulkerson algorithm with some edition. We will just add a source node to the beginning and a sink node at the end giving a capacity to every edge and calculating the maximum flow which is going to be the maximum cardinality bipartite matching.

```python
# Variables
INF = float('inf')
# Edge Class
class Edge:
    def __init__(self, back, front, capactiy):
        self.back = back
        self.front = front
        self.capacity = capactiy
        self.residual = None
        self.flow = 0
    def isResidual(self):
        return self.capacity == 0
    def remaining_capactiy(self):
        return self.capacity - self.flow
    def augment(self, bottleNeck):
        self.flow += bottleNeck
        self.residual.flow -= bottleNeck

class FlowNetwork:
    def __init__(self, n, source, sink):
        self.n = n
        self.source = source
        self.sink = sink
        self.graph = [[] for _ in range(n)]
        self.visited = [0] * n
        self.visitedToken = 1
        self.max_flow = 0

    def add_edge(self, back, front, capacity):
        edge = Edge(back, front, capacity)
        residual = Edge(front, back, 0)
        edge.residual = residual
        residual.residual = edge
        self.graph[back].append(edge)
        self.graph[front].append(residual)
    # Ford Fulkerson Algorithm
    def dfs(self, node, flow):
        if node == self.sink: return flow

        self.visited[node] = self.visitedToken
        edges = self.graph[node]
        for edge in edges:
            if edge.remaining_capactiy()>0 and self.visited[edge.front] != self.visitedToken:
                bottlneck = self.dfs(edge.front, min(flow, edge.remaining_capactiy()))
                if bottlneck >0:
                    edge.augment(bottlneck)
                    return bottlneck
        return 0

    def find_max_flow(self):
        f = self.dfs(self.source, INF)
        while f!=0:
            self.visitedToken += 1
            self.max_flow += f
            f = self.dfs(self.source, INF)
        return self.max_flow
```

Here's a small example for finding the MCBM:



We will use the Python code down below to outline the output of the algorithm on this graph:

```
#Application
ford = FlowNetwork(10, 0, 9)
# add edges from the source node 0 to the input nodes
ford.add_edge(0, 1, 1)
ford.add_edge(0, 2, 1)
ford.add_edge(0, 3, 1)
ford.add_edge(0, 4, 1)
# add edges of the graph
ford.add_edge(1, 5, 1)
ford.add_edge(1, 6, 1)
ford.add_edge(1, 8, 1)
ford.add_edge(2, 5, 1)
ford.add_edge(2, 8, 1)
ford.add_edge(3, 6, 1)
ford.add_edge(3, 7, 1)
ford.add_edge(4, 6, 1)
ford.add_edge(4, 7, 1)
ford.add_edge(4, 8, 1)
# add edges from output nodes to the sink node
ford.add_edge(5, 9, 1)
ford.add_edge(6, 9, 1)
ford.add_edge(7, 9, 1)
ford.add_edge(8, 9, 1)

mcbm = ford.find_max_flow()
print(mcbm)
```

The corresponding output is:

```
Python>> 4
```

Ford Fulkerson Network Flow

Definition:

The Edmonds-Karp algorithm is a specific implementation of the Ford-Fulkerson method for finding the maximum flow in a network flow graph. It efficiently solves the unspecific detail of choosing augmenting paths by employing a breadth-first search (BFS) strategy to find the shortest augmenting paths from the source to the sink. Unlike the original Ford-Fulkerson method that uses any augmenting path, the Edmonds-Karp algorithm ensures that the augmenting paths have the minimum number of edges, leading to a faster convergence.

The algorithm iteratively finds augmenting paths and updates the flow in the graph until no more augmenting paths can be found. It guarantees that the flow is non-decreasing at each step, and when no more augmenting paths are available, it produces the maximum flow.

By employing BFS to find augmenting paths, the Edmonds-Karp algorithm optimizes the search process by focusing on shorter paths first. This approach significantly improves the algorithm's performance, especially when the capacities of the edges are integral and relatively small. Due to its efficiency and guaranteed termination, the Edmonds-Karp algorithm has become one of the most widely used methods for solving network flow problems in various applications, including transportation, communication networks, and resource allocation...

The Edmonds-Karp algorithm reduces the time complexity of finding the maximum flow in a network flow graph from an unbounded $O(Ef)$ in the Ford-Fulkerson algorithm to $O(VE^2)$, where V is the number of vertices and E is the number of edges in the graph. The use of BFS to find shortest augmenting paths ensures that each augmenting path is of length at most V, and there can be at most E iterations to reach the maximum flow. This bounded complexity makes the Edmonds-Karp algorithm more efficient and practical for solving large-scale network flow problems compared to the unbounded complexity of the Ford-Fulkerson algorithm.

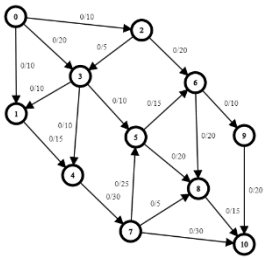
Algorithm:

```
from queue import Queue
# Variables
INF = float('inf')
# Edge Class
class Edge:
    def __init__(self, back, front, capacity):
        self.back = back
        self.front = front
        self.capacity = capacity
        self.residual = None
        self.flow = 0
    def isResidual(self):
        return self.capacity == 0
    def remaining_capacity(self):
        return self.capacity - self.flow
    def augment(self, bottleNeck):
        self.flow += bottleNeck
        self.residual.flow -= bottleNeck

class FlowNetwork:
    def __init__(self, n, source, sink):
        self.n = n
        self.source = source
        self.sink = sink
        self.graph = [[] for _ in range(n)]
        self.visited = [0] * n
        self.visitedToken = 1
        self.max_flow = 0

    def add_edge(self, back, front, capacity):
        edge = Edge(back, front, capacity)
        residual = Edge(front, back, 0)
        edge.residual = residual
        residual.residual = edge
        self.graph[back].append(edge)
        self.graph[front].append(residual)
    # Edmonds-Karp Algorithm
    def bfs(self):
        queue = Queue()
        self.visited[self.source] = self.visitedToken
        queue.put(self.source)
        prev = [None] * self.n
        while not queue.empty():
            node = queue.get()
            if node == self.sink: break
            for edge in self.graph[node]:
                if edge.remaining_capacity()>0 and self.visited[edge.front]!=self.visitedToken:
                    self.visited[edge.front] = self.visitedToken
                    prev[edge.front] = edge
                    queue.put(edge.front)
        if prev[self.sink] == None: return 0
        bottleneck = INF
        edge = prev[self.sink]
        while edge!=None:
            bottleneck = min(bottleneck, edge.remaining_capacity())
            edge = prev[edge.back]
        edge = prev[self.sink]
        while edge!=None:
            edge.augment(bottleneck)
            edge = prev[edge.back]
        return bottleneck

```

```
    def find_max_flow(self):
        f = self.bfs()
        while f!=0:
            # Mark all nodes as visited
            self.visitedToken += 1
            self.max_flow += f
            f = self.bfs()
        return self.max_flow

#Application
edmonds = FlowNetowrk…
print(edmonds.find_max_flow())
```

Example:

Here's a small example illustrating an example of input outputs for the Edmonds-Karp Algorithm:



We will use the Python code down below to outline the output of the algorithm on this graph:

```
from queue import Queue
# Variables
INF = float('inf')
# Edge Class
class Edge:
    def __init__(self, back, front, capacity):
        self.back = back
        self.front = front
        self.capacity = capacity
        self.residual = None
        self.flow = 0
    def isResidual(self):
        return self.capacity == 0
    def remaining_capacity(self):
        return self.capacity - self.flow
    def augment(self, bottleNeck):
        self.flow += bottleNeck
        self.residual.flow -= bottleNeck

class FlowNetwork:
    def __init__(self, n, source, sink):
        self.n = n
        self.source = source
        self.sink = sink
        self.graph = [[] for _ in range(n)]
        self.visited = [0] * n
        self.visitedToken = 1
        self.max_flow = 0

    def add_edge(self, back, front, capacity):
        edge = Edge(back, front, capacity)
        residual = Edge(front, back, 0)
```

```
        edge.residual = residual
        residual.residual = edge
        self.graph[back].append(edge)
        self.graph[front].append(residual)
    # Edmonds-Karp Algorithm
    def bfs(self):
        queue = Queue()
        self.visited[self.source] = self.visitedToken
        queue.put(self.source)
        prev = [None] * self.n
        while not queue.empty():
            node = queue.get()
            if node == self.sink: break
            for edge in self.graph[node]:
                if edge.remaining_capacity()>0 and self.visited[edge.front]!=self.visitedToken:
                    self.visited[edge.front] = self.visitedToken
                    prev[edge.front] = edge
                    queue.put(edge.front)
        if prev[self.sink] == None: return 0
        bottleneck = INF
        edge = prev[self.sink]
        while edge!=None:
            bottleneck = min(bottleneck, edge.remaining_capacity())
            edge = prev[edge.back]
        edge = prev[self.sink]
        while edge!=None:
            edge.augment(bottleneck)
            edge = prev[edge.back]
        return bottleneck

    def find_max_flow(self):
        f = self.bfs()
        while f!=0:
            # Mark all nodes as visited
            self.visitedToken += 1
            self.max_flow += f
            f = self.bfs()
        return self.max_flow

#Application
edmonds = FlowNetwork(11, 0, 10)
edmonds.add_edge(0, 2, 10)
edmonds.add_edge(0, 3, 20)
edmonds.add_edge(0, 1, 10)
edmonds.add_edge(1, 4, 15)
edmonds.add_edge(2, 6, 20)
edmonds.add_edge(2, 3, 5)
edmonds.add_edge(3, 5, 10)
edmonds.add_edge(3, 4, 10)
edmonds.add_edge(3, 1, 10)
edmonds.add_edge(4, 7, 30)
edmonds.add_edge(5, 6, 15)
edmonds.add_edge(5, 8, 20)
edmonds.add_edge(6, 9, 10)
edmonds.add_edge(6, 8, 20)
edmonds.add_edge(7, 5, 25)
edmonds.add_edge(7, 8, 5)
edmonds.add_edge(7, 10, 30)
edmonds.add_edge(8, 10, 15)
edmonds.add_edge(9, 10, 20)

print(edmonds.find_max_flow())
```

The corresponding output is:

```
Python>> 40
```

Capacity Scaling Algorithm

Definition:

Capacity scaling is an optimization technique used to improve the Ford-Fulkerson algorithm's performance. It aims to reduce the number of iterations required to find the maximum flow in a flow network by efficiently selecting augmenting paths. In the traditional Ford-Fulkerson algorithm, each iteration may consider paths with different flow values, leading to slow convergence. However, capacity scaling ensures that only augmenting paths with flow values that are multiples of a power of two are considered, significantly reducing the number of iterations.

The capacity scaling algorithm starts with an initial flow value of zero and gradually increases it by powers of two until it reaches the maximum flow. During each iteration, it identifies an augmenting path using a depth-first search or breadth-first search, but it restricts the flow along this path to multiples of the current scaling factor. This restriction ensures that the flow increases more rapidly, and each iteration can quickly find the path with the largest possible flow.

The time complexity of the capacity scaling algorithm is $O(E \log C)$, where E is the number of edges in the flow network, and C is the maximum capacity of any edge. This time complexity is much more efficient compared to the original Ford-Fulkerson algorithm, especially when the capacities are large. By exploiting the power of two scaling, capacity scaling efficiently converges to the maximum flow, making it a practical choice for solving maximum flow problems in large flow networks.

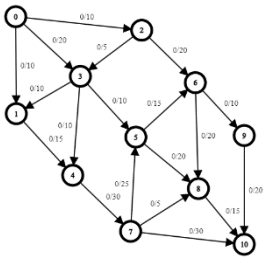
Algorithm:

```
# Variables
INF = float('inf')
# Edge Class
class Edge:
    def __init__(self, back, front, capacity):
        self.back = back
        self.front = front
        self.capacity = capacity
        self.residual = None
        self.flow = 0
    def isResidual(self):
        return self.capacity == 0
    def remaining_capacity(self):
        return self.capacity - self.flow
    def augment(self, bottleNeck):
        self.flow += bottleNeck
        self.residual.flow -= bottleNeck

```

```
class FlowNetwork:
    def __init__(self, n, source, sink):
        self.n = n
        self.source = source
        self.sink = sink
        self.graph = [[] for _ in range(n)]
        self.visited = [0] * n
        self.visitedToken = 1
        self.max_flow = 0
        self.delta = 0

    def add_edge(self, back, front, capacity):
        edge = Edge(back, front, capacity)
        residual = Edge(front, back, 0)
        edge.residual = residual
        residual.residual = edge
        self.graph[back].append(edge)
        self.graph[front].append(residual)
        self.delta = max(self.delta, capacity)
    # Cs-Karp Algorithm
    def dfs(self, node, flow):
        if node==self.sink: return flow
        self.visited[node] = self.visitedToken
        for edge in self.graph[node]:
            if (edge.remaining_capacity()>=self.delta and
                self.visited[edge.front] != self.visitedToken):
                bottleneck = self.dfs(edge.front, min(flow, edge.remaining_capacity()))
                if bottleneck > 0:
                    edge.augment(bottleneck)
                    return bottleneck
        return 0

    def find_max_flow(self):
        self.delta = 1<<(self.delta.bit_length()-1)
        while self.delta > 0:
            f = 1
            while f!=0:
                # Mark all nodes as visited
                self.visitedToken += 1
                f = self.dfs(self.source, INF)
                self.max_flow += f
            self.delta//=2
        return self.max_flow

#Application
cs = FlowNetwork…
print(cs.find_max_flow())
```

Example:

Here's a small example illustrating an example of input outputs for the Cs-Karp Algorithm:



We will use the Python code down below to outline the output of the algorithm on this graph:

```
# Variables
INF = float('inf')
# Edge Class
class Edge:
    def __init__(self, back, front, capacity):
        self.back = back
        self.front = front
        self.capacity = capacity
        self.residual = None
        self.flow = 0
    def isResidual(self):
        return self.capacity == 0
    def remaining_capacity(self):
        return self.capacity - self.flow
    def augment(self, bottleNeck):
        self.flow += bottleNeck
        self.residual.flow -= bottleNeck

class FlowNetwork:
    def __init__(self, n, source, sink):
        self.n = n
        self.source = source
        self.sink = sink
        self.graph = [[] for _ in range(n)]
        self.visited = [0] * n
        self.visitedToken = 1
        self.max_flow = 0
        self.delta = 0

    def add_edge(self, back, front, capacity):
        edge = Edge(back, front, capacity)
        residual = Edge(front, back, 0)
        edge.residual = residual
        residual.residual = edge
        self.graph[back].append(edge)
        self.graph[front].append(residual)
        self.delta = max(self.delta, capacity)
    # Capacity Scaling Algorithm
    def dfs(self, node, flow):
        if node==self.sink: return flow
        self.visited[node] = self.visitedToken
        for edge in self.graph[node]:
            if (edge.remaining_capacity()>=self.delta and
                self.visited[edge.front] != self.visitedToken):
                bottleneck = self.dfs(edge.front, min(flow, edge.remaining_capacity()))
                if bottleneck > 0:
                    edge.augment(bottleneck)
                    return bottleneck
        return 0

    def find_max_flow(self):
        self.delta = 1<<(self.delta.bit_length()-1)
        while self.delta > 0:
            f = 1
            while f!=0:
                # Mark all nodes as visited
                self.visitedToken += 1
                f = self.dfs(self.source, INF)
                self.max_flow += f
            self.delta//=2
        return self.max_flow

```

```
#Application
cs = FlowNetwork(11, 0, 10)
cs.add_edge(0, 2, 10)
cs.add_edge(0, 3, 20)
cs.add_edge(0, 1, 10)
cs.add_edge(1, 4, 15)
cs.add_edge(2, 6, 20)
cs.add_edge(2, 3, 5)
cs.add_edge(3, 5, 10)
cs.add_edge(3, 4, 10)
cs.add_edge(3, 1, 10)
cs.add_edge(4, 7, 30)
cs.add_edge(5, 6, 15)
cs.add_edge(5, 8, 20)
cs.add_edge(6, 9, 10)
cs.add_edge(6, 8, 20)
cs.add_edge(7, 5, 25)
cs.add_edge(7, 8, 5)
cs.add_edge(7, 10, 30)
cs.add_edge(8, 10, 15)
cs.add_edge(9, 10, 20)

print(cs.find_max_flow())
```

The corresponding output is:

```
Python>> 40
```

Dinic’s Algorithm

Definition:

Dinic's Algorithm is an efficient algorithm for solving the maximum flow problem in a flow network. It is an improvement over the Ford-Fulkerson algorithm and is based on the concept of layered graphs. The algorithm uses a series of level graphs to efficiently find augmenting paths, making it highly efficient in practice.

The main idea behind Dinic's Algorithm is to build a layered graph where each layer represents the distance from the source in terms of edge count. During the algorithm's execution, it explores these layers in a Breadth-First Search (BFS)-like manner, finding augmenting paths with increasing lengths. This way, the algorithm terminates when no more augmenting paths can be found in the layered graph.

One of the key features that make Dinic's Algorithm highly efficient is that it guarantees each phase of BFS to improve the distance of at least one node from the source. This results in the algorithm converging quickly, making it much faster compared to the classical Ford-Fulkerson algorithm.

The time complexity of Dinic's Algorithm is $O(V^2E)$ in the worst case, where V is the number of vertices and E is the number of edges in the flow network. However, in practice, the algorithm often performs much faster, especially in sparse graphs, due to the use of layered graphs and blocking flow optimization.

Due to its efficient performance, Dinic's Algorithm is widely used in real-world applications involving network flow problems, such as transportation, telecommunications, and resource allocation. Its ability to handle large-scale flow networks with excellent time complexity makes it a popular choice for solving maximum flow problems in practical scenarios.

Algorithm:

To implement Dinic's Algorithm, follow these steps:

1. Build Residual Graph: Create a residual graph from the original flow network. The residual graph represents the remaining capacity for each edge after the flow has been pushed through it. Initialize the flow of all edges to zero.

2. Construct Layered Graph: Use Breadth-First Search (BFS) to construct a layered graph that indicates the shortest path from the source to each node in terms of edge count. This helps in efficiently finding augmenting paths.

3. Find Blocking Flow: While there exists a blocking flow in the layered graph (an augmenting path from the source to the sink), find the minimum capacity (bottleneck) along the path. Then, push the maximum possible flow through this path, updating the residual graph accordingly.

4. Update Layered Graph: After each blocking flow is found, update the layered graph by re-running BFS to find the shortest paths. This ensures that the algorithm converges efficiently.

5. Repeat Steps 3 and 4: Continue finding blocking flows and updating the layered graph until no more augmenting paths can be found.

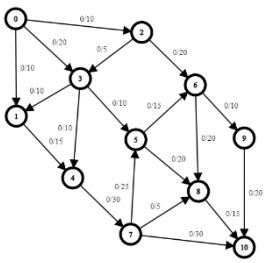
6. Calculate Maximum Flow: The maximum flow in the network is the total flow outgoing from the source. Once no more augmenting paths can be found, the algorithm terminates, and the maximum flow is obtained.

```
from queue import Queue
# Variables
INF = float('inf')
# Edge Class
class Edge:
    def __init__(self, back, front, capacity):
        self.back = back
        self.front = front
        self.capacity = capacity
        self.residual = None
        self.flow = 0
    def isResidual(self):
        return self.capacity == 0
    def remaining_capacity(self):
        return self.capacity - self.flow
    def augment(self, bottleNeck):
        self.flow += bottleNeck
        self.residual.flow -= bottleNeck

class FlowNetwork:
    def __init__(self, n, source, sink):
        self.n = n
        self.source = source
        self.sink = sink
        self.graph = [[] for _ in range(n)]
        self.visited = [0] * n
        self.visitedToken = 1
        self.max_flow = 0
        self.level = [0] * n

    def add_edge(self, back, front, capacity):
        edge = Edge(back, front, capacity)
        residual = Edge(front, back, 0)
        edge.residual = residual
        residual.residual = edge
        self.graph[back].append(edge)
        self.graph[front].append(residual)
    # Dinic' Algorithm
    def bfs(self):
        self.level = [-1] * self.n
        queue = Queue()
        queue.put(self.source)
        self.level[self.source] = 0
        while not queue.empty():
            node = queue.get()
            for edge in self.graph[node]:
                if edge.remaining_capacity()>0 and self.level[edge.front] == -1:
                    self.level[edge.front] = self.level[node] + 1
                    queue.put(edge.front)
        return self.level[self.sink]!=-1
```

```
51.
52.     def dfs(self, i, next, flow):
53.         if i == self.sink: return flow
54.         while next[i] < len(self.graph[i]):
55.             edge = self.graph[i][next[i]]
56.             if edge.remaining_capacity()>0 and self.level[edge.front] == self.level[i]+1:
57.                 bottleneck = self.dfs(edge.front, next, min(flow, edge.remaining_capacity()))
58.                 if bottleneck > 0:
59.                     edge.augment(bottleneck)
60.                     return bottleneck
61.             next[i] += 1
62.         return 0
63.
64.     def find_max_flow(self):
65.         while self.bfs():
66.             next = [0] * self.n
67.             f = self.dfs(self.source, next, INF)
68.             while f!=0:
69.                 self.max_flow += f
70.                 f = self.dfs(self.source, next, INF)
71.         return self.max_flow
72.
73. #Application
74. edmonds = FlowNetwork…
75.
76. print(edmonds.find_max_flow())
```

Example:

Here's a small example illustrating an example of input outputs for the Cs-Karp Algorithm:



We will use the Python code down below to outline the output of the algorithm on this graph:

```
 1. from queue import Queue
 2. # Variables
 3. INF = float('inf')
 4. # Edge Class
 5. class Edge:
 6.     def __init__(self, back, front, capacity):
 7.         self.back = back
 8.         self.front = front
 9.         self.capacity = capacity
10.         self.residual = None
11.         self.flow = 0
12.     def isResidual(self):
13.         return self.capacity == 0
14.     def remaining_capacity(self):
15.         return self.capacity - self.flow
16.     def augment(self, bottleNeck):
17.         self.flow += bottleNeck
18.         self.residual.flow -= bottleNeck
```

```
19.
20. class FlowNetwork:
21.     def __init__(self, n, source, sink):
22.         self.n = n
23.         self.source = source
24.         self.sink = sink
25.         self.graph = [[] for _ in range(n)]
26.         self.visited = [0] * n
27.         self.visitedToken = 1
28.         self.max_flow = 0
29.         self.level = [0] * n
30.
31.     def add_edge(self, back, front, capacity):
32.         edge = Edge(back, front, capacity)
33.         residual = Edge(front, back, 0)
34.         edge.residual = residual
35.         residual.residual = edge
36.         self.graph[back].append(edge)
37.         self.graph[front].append(residual)
38.     # Dinic' Algorithm
39.     def bfs(self):
40.         self.level = [-1] * self.n
41.         queue = Queue()
42.         queue.put(self.source)
43.         self.level[self.source] = 0
44.         while not queue.empty():
45.             node = queue.get()
46.             for edge in self.graph[node]:
47.                 if edge.remaining_capacity()>0 and self.level[edge.front] == -1:
48.                     self.level[edge.front] = self.level[node] + 1
49.                     queue.put(edge.front)
50.         return self.level[self.sink]!=-1
51.
52.     def dfs(self, i, next, flow):
53.         if i == self.sink: return flow
54.         while next[i] < len(self.graph[i]):
55.             edge = self.graph[i][next[i]]
56.             if edge.remaining_capacity()>0 and self.level[edge.front] == self.level[i]+1:
57.                 bottleneck = self.dfs(edge.front, next, min(flow, edge.remaining_capacity()))
58.                 if bottleneck > 0:
59.                     edge.augment(bottleneck)
60.                     return bottleneck
61.             next[i] += 1
62.         return 0
63.
64.     def find_max_flow(self):
65.         while self.bfs():
66.             next = [0] * self.n
67.             f = self.dfs(self.source, next, INF)
68.             while f!=0:
69.                 self.max_flow += f
70.                 f = self.dfs(self.source, next, INF)
71.         return self.max_flow
72.
73. #Application
74. edmonds = FlowNetwork(11, 0, 10)
75. edmonds.add_edge(0, 2, 10)
76. edmonds.add_edge(0, 3, 20)
77. edmonds.add_edge(0, 1, 10)
78. edmonds.add_edge(1, 4, 15)
79. edmonds.add_edge(2, 6, 20)
80. edmonds.add_edge(2, 3, 5)
81. edmonds.add_edge(3, 5, 10)
82. edmonds.add_edge(3, 4, 10)
83. edmonds.add_edge(3, 1, 10)
```

```
edmonds.add_edge(4, 7, 30)
edmonds.add_edge(5, 6, 15)
edmonds.add_edge(5, 8, 20)
edmonds.add_edge(6, 9, 10)
edmonds.add_edge(6, 8, 20)
edmonds.add_edge(7, 5, 25)
edmonds.add_edge(7, 8, 5)
edmonds.add_edge(7, 10, 30)
edmonds.add_edge(8, 10, 15)
edmonds.add_edge(9, 10, 20)

print(edmonds.find_max_flow())
```

The corresponding output is:

```
Python>> 40
```