

Graph implementation in many ways

Graphs can be implemented in several ways, depending on the specific requirements and use cases.

Here are some common implementations for graphs:

1. Adjacency Matrix:

In this implementation, a 2D matrix is used to represent the graph. The rows and columns of the matrix represent the vertices, and the values in the matrix indicate the presence or absence of edges between vertices. For an unweighted graph, a value of 1 or true can represent an edge, while a value of 0 or false represents no edge. For a weighted graph, the matrix can store the weight values instead of boolean values.

```
In [1]: class Graph:
        def __init__(self, num_vertices):
            self.num_vertices = num_vertices
            self.matrix = [[0] * num_vertices for _ in range(num_vertices)]

        def add_edge(self, source, destination):
            if 0 <= source < self.num_vertices and 0 <= destination < self.num_vertices:
                self.matrix[source][destination] = 1
                # Uncomment for weighted graph:
                # self.matrix[source][destination] = weight

        def remove_edge(self, source, destination):
            if 0 <= source < self.num_vertices and 0 <= destination < self.num_vertices:
                self.matrix[source][destination] = 0

        def print_graph(self):
            for row in self.matrix:
                print(row)

# Example usage
g = Graph(4)
g.add_edge(0, 1)
g.add_edge(0, 2)
g.add_edge(1, 3)
g.add_edge(2, 3)
g.print_graph()

executed in 46ms, finished 01:18:56 2023-08-08

[0, 1, 1, 0]
[0, 0, 0, 1]
[0, 0, 0, 1]
[0, 0, 0, 0]
```

2. Adjacency List:

This implementation uses an array of linked lists or dynamic arrays. Each element in the array represents a vertex, and the linked list or dynamic array associated with each vertex contains its adjacent vertices. This implementation is more memory-efficient than the adjacency matrix for sparse graphs (graphs with fewer edges).

```
In [7]: class Graph:
        def __init__(self):
            self.graph = {}

        def add_vertex(self, vertex):
            if vertex not in self.graph:
                self.graph[vertex] = []

        def add_edge(self, source, destination):
            if source in self.graph and destination in self.graph:
                self.graph[source].append(destination)
                self.graph[destination].append(source) # Uncomment for undirected graph

        def remove_vertex(self, vertex):
            if vertex in self.graph:
                del self.graph[vertex]
                for v in self.graph:
                    self.graph[v] = [x for x in self.graph[v] if x != vertex]

        def remove_edge(self, source, destination):
            if source in self.graph and destination in self.graph:
                self.graph[source] = [x for x in self.graph[source] if x != destination]
                self.graph[destination] = [x for x in self.graph[destination] if x != source] # Uncomment for undirected graph

        def get_vertices(self):
            return list(self.graph.keys())

        def get_edges(self):
            edges = []
            for vertex in self.graph:
                for neighbor in self.graph[vertex]:
                    edges.append((vertex, neighbor))
            return edges
```

Activate
Go to Set

```
# Example usage
g = Graph()
g.add_vertex('A')
g.add_vertex('B')
g.add_vertex('C')
g.add_edge('A', 'B')
g.add_edge('B', 'C')
g.add_edge('C', 'A')

print(g.get_vertices()) # Output: ['A', 'B', 'C']
print(g.get_edges()) # Output: [('A', 'B'), ('B', 'A'), ('B', 'C'), ('C', 'B'), ('C', 'A'), ('A', 'C')]
```

```
g.remove_edge('A', 'B')
g.remove_vertex('C')

print(g.get_vertices()) # Output: ['A', 'B']
print(g.get_edges()) # Output: []
```

executed in 24ms, finished 01:27:17 2023-08-08

```
['A', 'B', 'C']
[('A', 'B'), ('A', 'C'), ('B', 'A'), ('B', 'C'), ('C', 'B'), ('C', 'A')]
['A', 'B']
[]
```

3. Edge List:

This implementation maintains a list of edges in the graph. Each edge is represented as a pair of vertices or a triple (source vertex, destination vertex, weight). This implementation is simple and memory-efficient for storing the edges, but it may require additional processing for certain graph operations.

```
In [8]: class Graph:
        def __init__(self):
            self.edges = []

        def add_edge(self, source, destination):
            self.edges.append((source, destination))
            # Uncomment for weighted graph:
            # self.edges.append((source, destination, weight))

        def remove_edge(self, source, destination):
            self.edges = [(s, d) for s, d in self.edges if s != source or d != destination]

        def print_graph(self):
            for edge in self.edges:
                print(edge[0], "->", edge[1])

        # Example usage
        g = Graph()
        g.add_edge('A', 'B')
        g.add_edge('A', 'C')
        g.add_edge('B', 'D')
        g.add_edge('C', 'D')
        g.print_graph()
```

executed in 25ms, finished 01:34:43 2023-08-08

```
A -> B
A -> C
B -> D
C -> D
```

4. Incidence Matrix:

This implementation represents a graph as a 2D matrix where the rows represent vertices, and the columns represent edges. The matrix stores a value indicating the relationship between vertices and edges (e.g., -1 for the source vertex, 1 for the destination vertex, and 0 for unrelated vertices). This implementation is useful for certain graph algorithms, such as maximum flow algorithms.

```
In [10]: class Graph:
    def __init__(self, num_vertices, num_edges):
        self.num_vertices = num_vertices
        self.num_edges = num_edges
        self.matrix = [[0] * num_edges for _ in range(num_vertices)]

    def add_edge(self, source, destination, edge_index):
        if 0 <= source < self.num_vertices and 0 <= destination < self.num_vertices \
            and 0 <= edge_index < self.num_edges:
            self.matrix[source][edge_index] = -1
            self.matrix[destination][edge_index] = 1

    def remove_edge(self, edge_index):
        if 0 <= edge_index < self.num_edges:
            for i in range(self.num_vertices):
                self.matrix[i][edge_index] = 0

    def print_graph(self):
        for row in self.matrix:
            print(row)

# Example usage
g = Graph(4, 3)
g.add_edge(0, 1, 0)
g.add_edge(0, 2, 1)
g.add_edge(1, 3, 2)
g.add_edge(2, 3, 2)
g.print_graph()

executed in 16ms, finished 01:36:07 2023-06-08

[-1, -1, 0]
[1, 0, -1]
[0, 1, -1]
[0, 0, 1]
```

5. Compressed Sparse Row (CSR):

These examples demonstrate the implementation of a graph using an incidence matrix and compressed sparse row (CSR). The incidence matrix represents the graph as a matrix where rows represent vertices, and columns represent edges. The CSR implementation uses arrays to store the row indices and column indices of the edges efficiently.

```
In [11]: class Graph:
    def __init__(self, num_vertices, num_edges, edge_list):
        self.num_vertices = num_vertices
        self.num_edges = num_edges
        self.edge_list = edge_list
        self.row_indices = []
        self.column_indices = []

    def create_csr(self):
        for edge in self.edge_list:
            self.row_indices.append(edge[0])
            self.column_indices.append(edge[1])

    def print_graph(self):
        for i in range(self.num_edges):
            print(f"Row: {self.row_indices[i]}, Column: {self.column_indices[i]}")

# Example usage
g = Graph(4, 6, [(0, 1), (0, 2), (1, 0), (1, 3), (2, 3), (3, 2)])
g.create_csr()
g.print_graph()
```

executed in 22ms, finished 01:30:58 2023-08-08

```
Row: 0, Column: 1
Row: 0, Column: 2
Row: 1, Column: 0
Row: 1, Column: 3
Row: 2, Column: 3
Row: 3, Column: 2
```