**Math.inf – бесконечность**

**heapq — это стандартный модуль Python для работы с кучами. Куча — это специальная структура данных, которая реализует приоритетную очередь. В куче каждый элемент имеет приоритет, и элементы с наибольшим или наименьшим приоритетом (в зависимости от типа кучи) всегда находятся на вершине.**

**1. Инициализация графа (метод**\_\_init\_\_**):**

        self.num\_vertices = num\_vertices

        self.graph = [[math.inf] \* num\_vertices for \_ in range(num\_vertices)]

        for i in range(num\_vertices):

            self.graph[i][i] = 0

* Граф представляет собой **матрицу смежности**, где строки и столбцы соответствуют вершинам графа.
* При создании графа размерность матрицы num\_vertices x num\_vertices инициализируется значениями бесконечности (math.inf), что символизирует отсутствие пути между вершинами.
* Диагональные элементы матрицы устанавливаются в 0, так как расстояние от вершины до самой себя всегда равно нулю.

**2. Добавление ребра (метод**add\_edge**):**

def add\_edge(self, u: int, v: int, weight: float) -> None:

        """Добавляет ребро от вершины u к v с весом weight."""

        self.graph[u][v] = weight

        #добавить, если ориентированный

        #self.graph[v][u] = weight

* Этот метод добавляет ребро между вершинами u и v с весом weight. В графе, представленном матрицей смежности, это означает установку значения в матрице для ячейки [u][v].

**3. Вывод графа (метод**print\_graph**):**

    def print\_graph(self) -> None:

        """Выводит матрицу смежности графа в терминал."""

        print("Матрица смежности графа:")

        for row in self.graph:

            print(" ".join(str(i) if i != math.inf else "∞" for i in row))

* Этот метод выводит граф в виде матрицы смежности, заменяя значения math.inf (бесконечность) на символ "∞", чтобы представить отсутствие пути между вершинами.

**4. Сохранение графа в файл (метод**save\_to\_file**):**

    def save\_to\_file(self, filename: str) -> None:

        """Сохраняет граф в файл."""

        with open(filename, "w") as f:

            f.write(f"{self.num\_vertices}\n")

            for row in self.graph:

                f.write(" ".join(str(i) if i != math.inf else "∞" for i in row) + "\n")

* Этот метод сохраняет граф в файл. Он записывает количество вершин и саму матрицу смежности, преобразуя значения бесконечности в символ "∞".

**5. Загрузка графа из файла (метод**load\_from\_file**):**

    def load\_from\_file(self, filename: str) -> 'Graph':

        """Загружает граф из файла."""

        with open(filename, "r") as f:

            self.num\_vertices = int(f.readline().strip())

            self.graph = []

            for line in f:

                self.graph.append([math.inf if x == "∞" else float(x) for x in line.split()])

        return self

* Этот метод загружает граф из файла. Он сначала считывает количество вершин, затем для каждой строки в файле восстанавливает значения матрицы смежности, заменяя "∞" на math.inf.

**6. Алгоритм Дейкстры (метод**dijkstra**):**

def dijkstra(self, start: int) -> List[float]:

        """Алгоритм Дейкстры для поиска кратчайших путей от вершины start."""

        dist: List[float] = [math.inf] \* self.num\_vertices

        dist[start] = 0

        pq: List[tuple]  = [(0, start)]  # (расстояние, вершина)

        while pq:

            current\_dist, u = heapq.heappop(pq)

            if current\_dist > dist[u]:

                continue

            for v in range(self.num\_vertices):

                if self.graph[u][v] != math.inf:

                    weight = self.graph[u][v]

                    if dist[u] + weight < dist[v]:

                        dist[v] = dist[u] + weight

                        heapq.heappush(pq, (dist[v], v))

        return dist

* Алгоритм Дейкстры находит кратчайшие пути от заданной вершины (start) до всех остальных вершин в графе.
* Используется приоритетная очередь (с помощью heapq), чтобы всегда извлекать вершину с наименьшим текущим расстоянием и обновлять расстояния до соседних вершин.
* Возвращает список dist, где dist[i] — кратчайшее расстояние от start до вершины i.

**7. Алгоритм Форда-Беллмана (метод**bellman\_ford**):**

def bellman\_ford(self, start: int) -> Tuple[bool, List[float]]:

        """

        Алгоритм Форда-Беллмана для поиска кратчайших путей от вершины start.

        Возвращает (bool, List[float]), где bool — наличие отрицательного цикла.

        """

        dist = [math.inf] \* self.num\_vertices

        dist[start] = 0

        for \_ in range(self.num\_vertices - 1):

            for u in range(self.num\_vertices):

                for v in range(self.num\_vertices):

                    if self.graph[u][v] != math.inf and dist[u] + self.graph[u][v] < dist[v]:

                        dist[v] = dist[u] + self.graph[u][v]

        # Проверка на отрицательные циклы

        for u in range(self.num\_vertices):

            for v in range(self.num\_vertices):

                if self.graph[u][v] != math.inf and dist[u] + self.graph[u][v] < dist[v]:

                    return False, []  # Обнаружен отрицательный цикл

        return True, dist

* Алгоритм Форда-Беллмана находит кратчайшие пути от вершины start, но также способен обнаруживать отрицательные циклы.
* Он выполняет релаксацию всех рёбер графа на протяжении num\_vertices - 1 итераций.
* После этого проверяется наличие отрицательных циклов — если кратчайший путь можно еще уменьшить, значит, есть отрицательный цикл, и возвращается False.
* Возвращает кортеж, состоящий из булевого значения (наличие отрицательного цикла) и списка кратчайших расстояний.

