1. Какие представления графов Вы знаете?

**Матрица смежности (Adjacency Matrix)**

* **Описание**: Представляет граф с помощью квадратной матрицы, где строки и столбцы соответствуют вершинам, а элементы матрицы указывают наличие (или отсутствие) рёбер между вершинами.

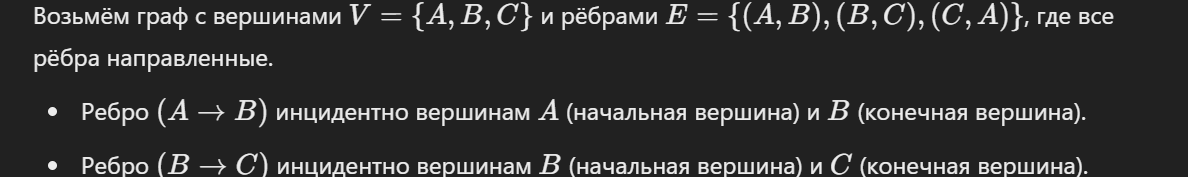
**Список смежности (Adjacency List)**

* **Описание**: Для каждой вершины создаётся список всех её соседей (вершин, с которыми она соединена рёбрами).
  + Например, для вершины viv\_ivi​ список содержит все вершины vjv\_jvj​, с которыми существует ребро.
  + Хранить для каждой вершины все ее соседние вершины

**Список рёбер (Edge List)**

* **Описание**: Это список всех рёбер графа, где каждое ребро представлено парой вершин (или тройкой вершин в случае направленных графов с указанием веса).

**Матрица инцидентности (Incidence Matrix)**

* **Описание**: Эта матрица представляет рёбра графа. В строках матрицы находятся вершины, а в столбцах — рёбра. Если вершина инцидентен ребру, то в соответствующей ячейке будет стоять 1 (или вес рёбер
* 

**Инцидентность** в теории графов — это отношение между вершинами и рёбрами графа, которое описывает, каким образом рёбра соединяют вершины. В других словах, инцидентность указывает, какие рёбра инцидентны (связаны с) определённой вершине.

1. В чем заключается поиск в ширину? Где рационально его использовать?

**Поиск в ширину (BFS, Breadth-First Search)** — это один из основных алгоритмов обхода графа, который проходит по всем вершинам графа, начиная с заданной вершины, и поочередно исследует все соседние вершины на одном уровне (в ширину), прежде чем перейти к вершинам на следующем уровне.

**Основные шаги алгоритма поиска в ширину:**

1. **Инициализация**:
   * Начинаем с выбранной стартовой вершины.
   * Помечаем её как посещённую.
   * Ставим её в очередь (обычно используется очередь FIFO — First In, First Out).
2. **Обработка очереди**:
   * Извлекаем вершину из очереди.
   * Для каждой непосещённой соседней вершины добавляем её в очередь и помечаем как посещённую.
3. **Завершение**:
   * Повторяем шаг 2, пока очередь не станет пустой.

**Описание работы алгоритма:**

1. Выбираем начальную вершину.
2. Помещаем её в очередь.
3. Пока очередь не пуста:
   * Извлекаем вершину из очереди.
   * Обрабатываем её (например, помечаем как посещённую).
   * Добавляем все её непосещённые соседние вершины в очередь.

**Когда рационально использовать поиск в ширину:**

1. **Нахождение кратчайшего пути в невзвешенных графах**:
   * В **невзвешенных** графах BFS используется для нахождения кратчайшего пути от начальной вершины до всех других. Алгоритм гарантирует, что, когда мы посещаем вершину, путь к ней будет минимальным.
2. **Поиск в графах с несколькими путями**:
   * Если нужно найти все возможные пути между двумя вершинами или просто исследовать все возможные вершины, BFS подходит идеально.
3. **Решение задач на уровне (например, в лабиринтах)**:
   * Если нужно пройти по всем точкам в лабиринте (или в любом другом пространстве, где важно проходить по уровням), BFS — отличный выбор.
4. **Использование в задачах маршрутизации**:
   * Например, для поиска маршрута в социальной сети, где нужно найти кратчайший путь от одного пользователя до другого, если в графе нет весов рёбер.
5. **Проверка связности графа**:
   * BFS можно использовать для проверки, является ли граф связным, то есть можно ли добраться от любой вершины к любой другой вершине.
6. **Решение задач на деревьях и графах с ограничениями на уровни**:
   * Например, если необходимо найти путь до вершины на заданном уровне или группе вершин на определённой глубине, BFS будет полезен.
7. В чем заключается поиск в глубину? В каких ситуациях рационально его использовать?

**Поиск в глубину (DFS, Depth-First Search)** — это алгоритм обхода графа, который начинает с какой-либо вершины и продвигается по графу, углубляясь в его ветви до тех пор, пока не достигнет вершины без непосещённых соседей. Затем он возвращается на один уровень вверх и продолжает обход в соседние вершины.

**Основные шаги алгоритма поиска в глубину:**

1. **Инициализация**:
   * Выбираем начальную вершину и помечаем её как посещённую.
   * Для каждого непосещённого соседа текущей вершины вызываем рекурсивный или итеративный обход вглубь.
2. **Обработка текущей вершины**:
   * Из текущей вершины переходим к непосещённым соседям, помечаем их как посещённые и продолжаем процесс углубления.
3. **Возврат**:
   * Если текущая вершина не имеет непосещённых соседей, то мы "возвращаемся" к предыдущей вершине (если она ещё не была полностью обработана) и продолжаем обход других её соседей.
   * Процесс продолжается до тех пор, пока все вершины не будут посещены.

**Когда рационально использовать поиск в глубину:**

1. **Поиск в деревьях и графах**:
   * DFS идеально подходит для обхода деревьев и графов, особенно когда нужно исследовать каждый путь до самого конца перед переходом к другому.
2. **Проверка связности графа**:
   * Если граф неориентированный, DFS может быть использован для проверки связности графа, то есть можно ли достичь любую вершину из любой другой.
3. **Поиск компонент связности**:
   * В неориентированных графах можно использовать DFS для нахождения компонент связности — групп вершин, которые соединены между собой.
4. **Поиск в лабиринтах**:
   * DFS подходит для решения задач, где нужно найти путь в лабиринте или сложных структурах, где важно пробовать все возможные пути, прежде чем вернуться назад.
5. **Проверка циклов в графах**:
   * DFS можно использовать для поиска циклов в графах, особенно в ориентированных. В процессе обхода, если мы встречаем вершину, которая уже находится в текущем пути (в стеке), то это означает, что цикл найден.
6. **Топологическая сортировка**:
   * В ориентированных ациклических графах (DAG) DFS используется для нахождения топологической сортировки, которая помогает упорядочить вершины так, чтобы для каждого ребра (u,v)(u, v)(u,v) вершина uuu шла перед вершиной vvv.
7. **Задачи, связанные с рекурсией или поиском всех путей**:
   * Когда нужно найти все возможные пути между двумя вершинами или пройти по всем возможным путям, DFS может быть использован для поиска этих путей.
8. **Поиск решений в задачах с несколькими шагами**:
   * DFS часто используется в задачах, где решение можно найти путём последовательных шагов, например, в задачах на нахождение пути или в головоломках (например, в решении судоку, поиске решения лабиринта).
9. В чем смысл топологической сортировки? Для чего она применяется?

**Топологическая сортировка** — это процесс упорядочивания вершин ориентированного ациклического графа (DAG, Directed Acyclic Graph) таким образом, что для каждого ребра (u,v)(u, v)(u,v) вершина uuu появляется перед вершиной vvv в списке сортировки. Другими словами, в топологической сортировке для каждого ребра графа соблюдается условие: если существует направленное ребро от вершины uuu к вершине vvv, то вершина uuu должна быть размещена раньше вершины vvv.

### Основные свойства топологической сортировки:

1. **Применимость**: Топологическая сортировка применяется только к **ориентированным ациклическим графам (DAG)**, то есть графам, в которых нет циклов. Если граф содержит цикл, то топологическая сортировка невозможна.
2. **Реализация**: Алгоритм топологической сортировки может быть реализован с помощью различных методов, включая поиск в глубину (DFS) или использование очереди для решения задачи в виде алгоритма Кхана.
3. **Уникальность**: Топологическая сортировка для графа может быть не единственной. Для некоторых графов может существовать несколько возможных порядков сортировки, но в любом случае соблюдается основное правило — вершины с ребрами, указывающими на них, всегда идут после вершины, от которой эти рёбра исходят.

**Применения топологической сортировки:**

1. **Упорядочивание задач**:
   * **Задачи с зависимостями**: Топологическая сортировка используется для упорядочивания задач, где выполнение одной задачи зависит от выполнения другой. Например, в системах сборки (например, Make) или в планировании проектов (например, в дифференцированном проектировании или на графах задач), где нужно сначала выполнить несколько подзадач, прежде чем переходить к следующей.
2. **Решение задач на графах**:
   * Топологическая сортировка используется в различных задачах на графах, таких как:
     + **Поиск максимального потока**: В некоторых алгоритмах, связанных с нахождением максимального потока, можно использовать топологическую сортировку для упорядочивания вершин.
     + **Алгоритмы на графах с зависимостями**: Например, для решения задач, связанных с вычислением маршрутов в графах с зависимыми путями.
3. **Упорядочивание вычислений в вычислительных системах**:
   * В распределённых вычислениях или в вычислительных графах часто используется топологическая сортировка для определения порядка вычислений. Это особенно важно при выполнении вычислений, которые могут быть параллельными, но должны соблюдаться зависимости.
4. **Поиск решения в задачах с зависимостями**:
   * Применяется в анализе зависимостей, например, для обработки зависимостей в базе данных, в компиляторах при анализе зависимостей между операциями и переменными, для нахождения порядка выполнения команд.
5. **Алгоритмы на графах с назначенными приоритетами**:
   * В таких случаях топологическая сортировка помогает установить порядок действий, где некоторые задачи должны быть выполнены до других.
6. Что такое минимальное остовное дерево?

**Минимальное остовное дерево (MST, Minimum Spanning Tree)** — это подмножество рёбер связанного взвешенного графа, которое соединяет все вершины графа и минимизирует сумму весов рёбер. Важно, что минимальное остовное дерево должно быть связным (содержать все вершины) и не содержать циклов (это дерево).

**Основные свойства минимального остовного дерева:**

1. **Минимизация веса**: Общее количество весов рёбер в минимальном остовном дереве является наименьшим среди всех возможных остовных деревьев графа.
2. **Связанность**: Остовное дерево должно включать все вершины графа, но только n−1n-1n−1 рёбер, где nnn — количество вершин. Это минимально возможное количество рёбер для связности графа.
3. **Отсутствие циклов**: Остовное дерево является деревом, то есть в нём нет циклов.
4. В чем заключается стандартный алгоритм построения минимального остовного дерева?

Стандартные алгоритмы построения **минимального остовного дерева** (MST) включают два наиболее известных метода: **алгоритм Прима** и **алгоритм Краскала**. Оба алгоритма жадные, то есть они строят решение шаг за шагом, выбирая на каждом шаге оптимальный локальный выбор.

### 1. ****Алгоритм Краскала**** (Kruskal's Algorithm):

**Описание**: Алгоритм Краскала работает жадным методом, начиная с рёбер минимального веса и добавляя их в остовное дерево, при этом избегая формирования циклов. Алгоритм использует структуру данных **Union-Find** (или **Disjoint Set Union, DSU**) для отслеживания, какие вершины уже соединены, и для предотвращения образования циклов.

#### Шаги алгоритма Краскала:

1. **Сортировка рёбер**:
   * Все рёбра графа сортируются по весу в порядке возрастания.
2. **Инициализация**:
   * Инициализируем структуру данных Union-Find, чтобы отслеживать компоненты связности (множества вершин).
   * Каждая вершина в начале является отдельной компонентой.
3. **Обработка рёбер**:
   * Проходим по отсортированным рёбрам и для каждого рёбра (u,v)(u, v)(u,v) проверяем, принадлежат ли вершины uuu и vvv одной компоненте (используя Union-Find).
   * Если вершины uuu и vvv не принадлежат одной компоненте, добавляем ребро (u,v)(u, v)(u,v) в минимальное остовное дерево и объединяем эти компоненты.
   * Если вершины уже соединены (при добавлении ребра возникает цикл), пропускаем это ребро.
4. **Завершение**:
   * Алгоритм завершится, когда количество рёбер в остовном дереве станет равно n−1n - 1n−1 (где nnn — количество вершин), или когда все рёбра будут рассмотрены.

### ****Алгоритм Прима**** (Prim's Algorithm):

**Описание**: Алгоритм Прима также работает жадным методом, но он строит остовное дерево, начиная с одной вершины и постепенно добавляя к дереву вершины, которые соединены с деревом минимальными рёбрами.

#### Шаги алгоритма Прима:

1. **Инициализация**:
   * Выбираем произвольную вершину как начальную.
   * Инициализируем список рёбер, которые будут добавлены в дерево (начинаем с рёбер, соединяющих начальную вершину).
   * Используем структуру данных (например, кучу или приоритетную очередь), чтобы поддерживать рёбра с минимальным весом, которые соединяют уже добавленные вершины с остальными вершинами.
2. **Обработка рёбер**:
   * На каждом шаге выбираем ребро с минимальным весом, которое соединяет вершину из уже построенного дерева с вершиной из оставшихся вершин.
   * Добавляем это ребро в остовное дерево.
   * Обновляем список рёбер, которые соединяют новые вершины с остальными вершинами.
3. **Завершение**:
   * Алгоритм завершится, когда все вершины будут включены в остовное дерево, и количество рёбер будет равно n−1n - 1n−1.

**Преимущества алгоритма Прима**:

1. К какой категории алгоритмов относятся алгоритмы Прима и Крускала?

**Алгоритмы Прима и Краскала** относятся к категории **жадных алгоритмов** (или **грейдиентных алгоритмов**, от англ. **greedy algorithms**).

**Жадные алгоритмы:**

Жадные алгоритмы решают задачу путем последовательного выбора локально оптимальных решений, которые, как предполагается, приведут к глобально оптимальному решению. В отличие от других типов алгоритмов, жадные алгоритмы делают "жадный" выбор на каждом шаге, не задумываясь о последствиях выбора на будущем.

**Алгоритм Прима и Краскала как жадные алгоритмы:**

1. **Алгоритм Прима**:
   * В алгоритме Прима на каждом шаге выбирается ребро с минимальным весом, которое соединяет уже построенное остовное дерево с ещё не включёнными в него вершинами. Таким образом, каждый шаг является "жадным выбором" минимального ребра.
2. **Алгоритм Краскала**:
   * В алгоритме Краскала на каждом шаге выбирается ребро с минимальным весом, которое не образует цикл, и добавляется в остовное дерево. Это также является "жадным выбором" минимального ребра.

**Почему они жадные:**

* **Локально оптимальные решения**: Оба алгоритма делают выбор на каждом шаге, выбирая ребро с минимальным весом, что является локально оптимальным (минимизация веса на каждом шаге).
* **Не учитывают глобальную картину**: Они не планируют весь путь до конца, а просто выбирают оптимальное решение на каждом шаге, что характерно для жадных алгоритмов.

1. Опишите один шаг алгоритма Крускала? Когда алгоритм прекращает свою работу?

**Один шаг алгоритма Краскала**

Алгоритм Краскала работает жадным методом, постепенно строя минимальное остовное дерево, начиная с рёбер минимального веса и добавляя их, если они не образуют цикл. На каждом шаге алгоритм выполняет следующие действия:

1. **Выбор минимального ребра**:
   * Алгоритм выбирает ребро с минимальным весом из всех рёбер графа, которые ещё не были рассмотрены. Рёбра предварительно сортируются по возрастанию веса.
2. **Проверка наличия цикла**:
   * Для выбранного ребра проверяется, не образует ли оно цикл в уже построенном остовном дереве. Это делается с помощью структуры данных **Union-Find** (или **Disjoint Set Union**, DSU), которая отслеживает компоненты связности.
   * Если обе вершины ребра принадлежат одной компоненте (то есть они уже соединены в дереве), то добавление этого рёбра приведёт к образованию цикла, и оно пропускается.
   * Если вершины принадлежат разным компонентам, то ребро добавляется в остовное дерево, и компоненты объединяются.
3. **Обновление состояния**:
   * После добавления ребра в остовное дерево обновляются компоненты связности. Если ребро добавляется, то в структуре Union-Find объединяются компоненты, в которые входят вершины этого ребра.

# 1. Выбираем минимальное ребро (u, v) из всех рёбер.

edge = min(edge\_list)

# 2. Проверяем, образует ли ребро цикл:

if find(u) != find(v): # Рёбра принадлежат разным компонентам

# 3. Добавляем ребро в остовное дерево.

MST.append(edge)

# 4. Объединяем компоненты.

union(u, v)

**Когда алгоритм прекращает свою работу?**

Алгоритм Краскала прекращает свою работу, когда выполнены следующие условия:

* **Количество рёбер в остовном дереве становится равным n−1n - 1n−1**, где nnn — количество вершин в графе. Это минимальное количество рёбер для связности графа, и при его достижении все вершины будут соединены.
* Или **все рёбра графа были рассмотрены**, но если граф не был связным, то не удастся получить остовное дерево (оно не существует).