­ МИНИСТЕРСТВО ОБРАЗОВАНИЯ РЕСПУБЛИКИ БЕЛАРУСЬ

Учреждение образования

«БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНОЛОГИЧЕСКИЙ

УНИВЕРСИТЕТ»

ПОИСК В ГЛУБИНУ И ПОИСК В ШИРИНУ

Научная секция «Основы алгоритмизации и программирования»

Подшиваленко Диана Игоревна,

студентка I курса

факультета информационных технологий

Научный руководитель:

Белодед Николай Иванович,

доцент, кандидат технических наук БГТУ

Минск, 2023

**Реферат**

Работа \_\_ с., 2 ч., \_\_ рис., \_\_ табл., \_\_ источников, 4 прил. ГРАФ, АЛГОРИТМ, ПОИСК, МЕТОД, ПОИСК В ШИРИНУ, ПОИСК В ГЛУБИНУ,

Объектом исследования являются методы работы с графами.

Цель работы – теоретическое исследование методов поиска в глубину и ширину; применение результатов теоретического исследования для реализации данных методов с использованием языка C++.

По результатам теоретических исследований, используя язык программирования С++, были реализованы алгоритмы поиска в ширину и в глубину.

Реализованные в С++ алгоритмы помогают понять сущность методов обработки графов, таких как – поиск в ширину и в глубину.

Содержание

|  |  |
| --- | --- |
| Введение………………………………………………………………………………… |  |
| 1 Аналитический обзор методов поиска в графах……………………………………   * 1. Поиск в глубину……………………………………………………………….   1.2 Поиск в ширину……………………………………………………………….  2 Реализация алгоритмов поиска в глубину и в ширину…………………………….  2.1 Реализация алгоритма DFS для обхода графа на языке С++……………….  2.2 Реализация алгоритма BFS для обхода графа на языке С++……………….  Заключение………………………………………………………………………………  Приложение А…………………………………………………………………………...  Приложение Б…………………………………………………………………………...  Приложение В…………………………………………………………………………...  Приложение Г…………………………………………………………………………. |  |

**Введение**

Проблема решения задач, в которых используется граф, является одной из наиболее актуальных и сложных для многих областей науки и технологий. Графы являются мощным инструментом для моделирования и анализа связей и взаимосвязей между объектами, их структурой и характеристиками. Они находят применение во множестве дисциплин, включая информатику, теорию вероятностей, экономику, социологию, химию и другие [].

Решение задач, в которых используется граф, вызывает ряд сложностей:

1. Сама по себе структура графа может быть довольно сложной и содержать большое количество узлов и связей. Это может существенно затрудняет работу с графом и требует применения специальных алгоритмов и методов для его анализа и обработки;
2. Решение задач на графах может предполагать нахождение оптимального пути или пути с определенными характеристиками. Это требует применения алгоритмов поиска пути, таких как алгоритм Дейкстры или алгоритмы поиска в глубину и ширину. Эти алгоритмы могут потребовать значительных вычислительных ресурсов и времени, особенно при работе с большими графами;

3) Задачи на графах могут быть связаны с оптимизацией и нахождением оптимальных решений. Это является NP-полной задачей, что означает, что ее решение может занимать значительное время, особенно при работе с большими графами.

Все эти факторы делают решение задач на графах сложным и требующим специализированных навыков и знаний. Поэтому, для эффективного решения задач, связанных с использованием графа, необходимо изучение и применение передовых методов и алгоритмов, а также использование современных вычислительных технологий.

Решение задач на графах представляет огромный потенциал и имеет значительное значение для различных областей науки и технологий. Благодаря графам мы можем получать более глубокое понимание сложных систем и структур, принимать более обоснованные решения и улучшать производительность и эффективность в различных областях деятельности.

В настоящей работе мы поставили цели: провести теоретическое исследование методов поиска в глубину и ширину; реализовать методы поиска в глубину и ширину с использованием языка C++.

Задачи:

- получить общее представление о поиске в глубину и ширину;

- написать программы их реализации на языке программирования высокого уровня C++.

**1 Аналитический обзор методов поиска в графах**

**1.1** **Поиск в глубину**

DFS, или Depth First Search, – поиск в глубину, позволяющий найти маршрут от точки A до точки B. Используется в графах – особых структурах, состоящих из точек-вершин и ребер-путей. DFS ищет маршрут по графу «в глубину»: на каждом шаге «уходит» все дальше. Общая идея алгоритма состоит в следующем: для каждой не пройденной вершины необходимо найти все не пройденные смежные вершины и повторить поиск для них. Один из вариантов использования этого алгоритма – это поиск выхода из лабиринта. Поиск в глубину начинает работу в заданной точки, на каждом шаге проходит до следующего поворота и выбирает направление. Если мы пришли в тупик, то он возвращается к предыдущему повороту и идет в другую сторону. В результате мы найдем правильный путь при условии, что он вообще существует.

Если существует несколько путей, то DFS найдет хотя бы один из них.

Также данный алгоритм нужен для решения следующих задач:

- выделение компонент связности;

- проверка графа на двудольность;

- поиск цикла в ориентированном графе;

- топологическая сортировка;

- поиск мостов.

Рассмотрим алгоритм работы DFS. Пусть мы сейчас находимся в какой-то из вершин графа. Тогда последовательность действий, будет следующая:

- пойти в какую-нибудь смежную вершину, непосещенную ранее;

- запустить из этой вершины поиск в глубину;

- если все смежные вершины посещены, то вернуться назад;

- повторить пункты 1-3 для вершин, которые не были посещены.

Рассмотрим это на конкретном примере. Пусть у нас имеется следующий граф (Рисунок 1.1) (непосещенные вершины будут показаны синим цветом):

Рисунок 1.1 – Заданный граф

Запустим поиск в глубину для первой вершины. Сначала на вход функции поступает вершина с номером 1. Помечаем ее как посещенную (Рисунок 1.2) (на рисунке посещенные вершины будет окрашены в голубой цвет).

Рисунок 1.2 – Граф на первом шаге

Далее берем первую непосещенную вершину из списка смежности вершины с номером 1. Пусть это будет вершина с номером 2. Запускаем поиск в глубину от нее. Она помечается как посещенная (Рисунок 1.3).

Рисунок 1.3 – Граф на втором шаге

У вершины с номером 2 есть непосещенный сосед с номером 4, значит перемещаемся туда. Запускаем поиск в глубину от вершины 4. Помечаем ее как посещенную (Рисунок 1.4).

Рисунок 1.4 – Граф на третьем шаге

У вершины с номером 4 есть только сосед с номером 2, но он уже посещен. Значит DFS от вершины с номером 4 завершает свою работу и возвращается на предыдущий уровень, то есть в вершину с номером 2 (вершины, у которых все соседи посещены будем обозначать серым цветом) (Рисунок 1.5).

Рисунок 1.5 – Граф на четвертом шаге

У вершины с номером 2 больше нет непосещенных соседей, поэтому тоже окрашиваем ее в серый цвет и возвращаемся в вершину с номером 1 (Рисунок 1.6).

Рисунок 1.6 – Граф на пятом шаге

У вершины с номером 1 еще есть непосещенные соседи. Это вершина с номером 3 и с номером 5. Пойдем, например, в вершину с номером 3. Запускается DFS от вершины с номером 3. Она помечается как посещенная (Рисунок 1.7).

Рисунок 1.7 – Граф на шестом шаге

У нее есть непосещенный сосед с номером 6. Переходим в него и запускаем DFS. Вершина с номером 6 помечается как посещенная (Рисунок 1.8).

Рисунок 1.8 – Граф на седьмом шаге

У нее есть непосещенная вершина с номером 5. Переходим в нее и запускаем DFS. Помечаем ее как посещенную (Рисунок 1.9).

Рисунок 1.9 – Граф на восьмом шаге

У вершины с номером 5 нет непосещенных соседей, значит DFS от нее завершается и возвращается в вершину с номером 6 (вершину с номером 5 окрашиваем в серый цвет) (Рисунок 1.10).

Рисунок 1.10 – Граф на девятом шаге

У вершины с номером 6 тоже нет непосещенных соседей, значит DFS от нее завершается и возвращается в вершину с номером 3 (вершину с номером 6 окрашиваем в серый цвет) (Рисунок 1.11).

Рисунок 1.11 – Граф на десятом шаге

Далее у вершины с номером 3 нет непосещенных соседей, значит DFS от нее завершается и возвращается в вершину с номером 1 (вершину с номером 3 окрашиваем в серый цвет).

Рисунок 1.12 – Граф на одиннадцатом шаге

Мы вернулись в вершину с номером 1, и у нее уже не осталось непосещенных соседей, поэтому DFS завершает работу (Рисунок 1.13)

Рисунок 1.13 – Граф на двенадцатом шаге

Как мы видим, наш алгоритм успешно обошел весь граф от нашей стартовой вершины с номером 1.

**1.2 Поиск в ширину**

BFS, или Breadth First Search – алгоритм обхода графа в ширину. BFS осуществляет обход графа по уровням: сначала проходит по всем ближайшим от начальной точки вершинам, потом спускается глубже. Выглядит это так: алгоритм начинает в заранее выбранной вершине и сначала «посещает» и отмечает всех соседей этой вершины. Потом он переходит к соседям посещенных вершин, затем продолжает по тому же принципу. Из-за характера распространения, похожего на волну, алгоритм еще называют волновым. BFS – один из двух популярных алгоритмов обхода. Алгоритм поиска в ширину работает как на ориентированных, так и на неориентированных графах.

Данный алгоритм используют для решения следующих задач:

- поиска оптимального пути;

- поиска компонент связности;

- поиск решения с минимальным количеством ходов;

- для оптимизации памяти при обходе графа в некоторых ситуациях;

- для работы с информацией в определенных структурах данных, таких как деревья.

Рассмотрим алгоритм работы BFS. Пусть мы сейчас находимся в какой-то из вершин графа. Тогда последовательность действий, будет следующая:

- добавить всех непосещенных соседей нашей вершины в очередь и указать расстояние до них;

- пока очередь не пуста, достаем первую вершину из очереди;

- добавляем всех непосещенных соседей этой вершины в очередь, указываем расстояния до них и переходим к следующей.

Рассмотрим это на конкретном примере. Пусть у нас имеется следующий граф (Рисунок 1.14).

**∞**

**∞**

**∞**

**∞**

**∞**

**∞**

**∞**

**Очередь**

Рисунок 1.14 – Заданный граф

Запускаем поиск в ширину для первой вершины. Изначально кратчайшие пути для каждой вершины из первой как бы не существуют (для обозначения будем использовать ∞). При запуске функции на вход поступает вершина с номером 1. Для хранения вершин, которые ожидают обработку, будем использовать очередь. Когда поступила первая вершина, указываем расстояние кратчайшего пути до нее равное 0 и добавляем ее в нашу очередь (окончательные расстояния будем помечать черным цветом, а промежуточные будут окрашиваться серым) (Рисунок 1.15).

**0**

**∞**

**∞**

**∞**

**∞**

**∞**

**∞**

**Очередь**

Рисунок 1.15 – Граф на первом шаге

Затем, пока очередь не пуста, извлекаем вершину из очереди. Когда это происходит, то можно считать расстояние до вершины окончательным (помечаем его черным цветом). Теперь нужно просмотреть всех соседей вершины 1. Это вершины с номером 2, 3, 5. До них пока указываем расстояние равное 1 и помещаем их в очередь (Рисунок 1.16).

**0**

**1**

**1**

**∞**

**1**

**∞**

**∞**

**Очередь**

Рисунок 1.16 – Граф на втором шаге

Стартовая вершина обработана. Переходим к следующей вершине с номером 2. Извлекаем ее из очереди и помечаем 1 как окончательное расстояние до нее (Рисунок 1.17).

**0**

**1**

**1**

**∞**

**1**

**∞**

**∞**

**Очередь**

Рисунок 1.17 – Граф на третьем шаге

Далее добавляем в очередь всех соседей вершины 2. Вершину с номером 1 уже обработали, а вот вершина с номером 4 еще не встречалась. Поэтому указываем расстояние до нее 2 и помещаем в очередь. Извлекаем следующую вершину с номером 5 (Рисунок 1.18).

**0**

**1**

**1**

**2**

**1**

**∞**

**∞**

**Очередь**

Рисунок 1.18 – Граф на четвертом шаге

Помечаем расстояние до нее как окончательное и добавляем еще не обработанных соседей этой вершины в очередь. Это вершины с номерами 6 и 7. Указываем расстояние до них равное 2. Извлекаем следующую вершину с номером 3 (Рисунок 1.19).

**Очередь**

**0**

**1**

**1**

**2**

**1**

**2**

**2**

Рисунок 1.19 – Граф на пятом шаге

Помечаем расстояние до нее как окончательное. У нее не осталось непосещенных соседей, поэтому переходим к следующей вершине в нашей очереди и извлекаем ее. Это вершина с номером 4 (Рисунок 1.20).

**0**

**1**

**1**

**2**

**1**

**2**

**2**

**Очередь**

Рисунок 1.20 – Граф на шестом шаге

Помечаем расстояние как окончательное. У нее тоже не осталось непосещенных соседей, поэтому опять переходим к следующей вершине в очереди. Извлекаем вершину с номером 6 (Рисунок 1.21).

Помечаем расстояние как окончательное. Непосещенных соседок снова не осталось, переходим к следующей вершине в очереди. Это вершина с номером 7. Извлекаем ее (Рисунок 1.22).

**0**

**1**

**1**

**2**

**1**

**2**

**2**

**Очередь**

Рисунок 1.21 – Граф на седьмом шаге

**0**

**1**

**1**

**2**

**1**

**2**

**2**

**Очередь**

Рисунок 1.22 – Граф на восьмом шаге

Помечаем расстояние как окончательное. Соседок, которых не посетили, не осталось, и наша очередь пуста. Алгоритм завершает работу (Рисунок 1.23)

**0**

**1**

**1**

**2**

**1**

**2**

**2**

**Очередь**

Рисунок 1.23 – Граф на девятом шаге

Как видим, алгоритм успешно нашел кратчайшие пути до других вершин от нашей стартовой вершины с номером 1.

**1.3 Алгоритм Дейкстры**

Алгоритм Дейкстры – это метод, который находит кратчайший путь от одной вершины графа к другой. Граф – структура из точек-вершин, соединенных ребрами-отрезками. Ребра – это связи, по ним можно двигаться от одной вершины к другой.

Алгоритм Дейкстры работает для графов, у которых нет ребер с отрицательным весом, т.е. таких, при прохождении через которые длина пути как бы уменьшается.

В отличие от похожих методов, алгоритм Дейкстры ищет оптимальный маршрут от одной заданной вершины ко всем остальным. Попутно он высчитывает длину пути – суммарный вес ребер, по которым проходит при этом маршруте.

Основная задача – поиск кратчайшего пути по схеме, где множество точек соединено между собой отрезками. Практических примеров использования алгоритма много:

­– автоматическое построение маршрута на онлайн-карте;

– поиск системой бронирования наиболее быстрых или дешевых билетов, в том числе с возможными пересадками;

– моделирование движения робота, который перемещается по местности;

– разработка поведения неигровых персонажей, создание игрового ИИ в геймдеве;

– автоматическая обработка транспортных потоков;

– маршрутизация движения данных в компьютерной сети;

– расчет движения тока по электрическим цепям.

Рассмотрим алгоритм Дейкстры в работе на конкретном примере. Пусть у нас имеется следующий граф (Рисунок 1.24).

**0**

**∞**

**∞**

**∞**

**∞**

**∞**

**∞**

**1**

**3**

**4**

**2**

**10**

**7**

**1**

**3**

Рисунок 1.24 – Заданный граф

Пусть нужно найти кратчайшие расстояния от 1 вершины до остальных. Будем использовать следующие обозначения: непосещенная вершина – синий цвет, посещенная вершина – серый цвет, цифры красного цвета рядом с ребрами – расстояние между вершинами, цифры черного цвета над вершинами – текущее кратчайшее расстояние до этой вершины из 1. Из вершины 1 в вершину 1 кратчайшее расстояние помечаем равным 0. До всех остальных вершин предполагается, что пути нет. Для этого будем использовать символ бесконечности. Помечаем 1 вершину как посещенную (серый цвет) и просматриваем всех ее соседей для поиска текущих кратчайших расстояний до них. Для этого сравниваем текущее кратчайшее расстояние соседки с расстоянием до текущей вершины + расстояние между данными вершинами. Для вершины 1 соседки будут вершины с номерами 2, 3, 5. Тогда текущее кратчайшее расстояние пока что не существует (символ бесконечности). Проверяем: текущее кратчайшее расстояние больше расстояния до вершины 1(0) + расстояние между вершинами 1 и 2 (1). Да. Тогда текущее кратчайшее расстояние до 2 вершины становится 0 + 1 = 1. Аналогичные действия делаем для вершины 3 и 5. (Рисунок 1.25).

**0**

**1**

**7**

**∞**

**10**

**∞**

**∞**

**1**

**3**

**4**

**2**

**10**

**7**

**1**

**3**

Рисунок 1.25 – Граф на первом шаге

Далее, когда все соседи 1 вершины посещены ищем ближайшую непосещенную вершину. Это 2 вершина. Помечаем ее как посещенную. И находим кратчайшие расстояния до ее соседей таким же образом, как это делали для соседей 1 вершины. (Рисунок 1.26).

**0**

**1**

**7**

**4**

**10**

**∞**

**∞**

**1**

**3**

**4**

**2**

**10**

**7**

**1**

**3**

Рисунок 1.26 – Граф на втором шаге

Далее, когда все соседи 2 вершины посещены ищем ближайшую непосещенную вершину. Это 4 вершина. Помечаем ее как посещенную. И находим кратчайшие расстояния до ее соседей таким же образом, как это делали для предыдущих вершин. (Рисунок 1.27).

**0**

**1**

**7**

**4**

**10**

**8**

**∞**

**1**

**3**

**4**

**2**

**10**

**7**

**1**

**3**

Рисунок 1.27 – Граф на третьем шаге

Затем, когда все соседи 4 вершины посещены ищем ближайшую непосещенную вершину. Это 3 вершина. Помечаем ее как посещенную. И находим кратчайшие расстояния до ее соседей таким же образом, как это делали для предыдущих вершин. (Рисунок 1.28).

**0**

**1**

**7**

**4**

**10**

**8**

**10**

**1**

**3**

**4**

**2**

**10**

**7**

**1**

**3**

Рисунок 1.28 – Граф на четвертом шаге

Затем, когда все соседи 3 вершины посещены ищем ближайшую непосещенную вершину. Это 7 вершина. Помечаем ее как посещенную. И находим кратчайшие расстояния до ее соседей таким же образом, как это делали для предыдущих вершин. (Рисунок 1.29).

**0**

**1**

**7**

**4**

**10**

**8**

**10**

**1**

**3**

**4**

**2**

**10**

**7**

**1**

**3**

Рисунок 1.29 – Граф на пятом шаге

Затем, когда все соседи 7 вершины посещены ищем ближайшую непосещенную вершину. Это 5 вершина. Помечаем ее как посещенную. И находим кратчайшие расстояния до ее соседей таким же образом, как это делали для предыдущих вершин. (Рисунок 1.30).

Затем, когда все соседи 5 вершины посещены ищем ближайшую непосещенную вершину. Это 6 вершина. Помечаем ее как посещенную. И находим кратчайшие расстояния до ее соседей таким же образом, как это делали для предыдущих вершин. (Рисунок 1.31).

**0**

**1**

**7**

**4**

**10**

**8**

**10**

**1**

**3**

**4**

**2**

**10**

**7**

**1**

**3**

Рисунок 1.30 – Граф на шестом шаге

**0**

**1**

**7**

**4**

**10**

**8**

**10**

**1**

**3**

**4**

**2**

**10**

**7**

**1**

**3**

Рисунок 1.31 – Граф на седьмом шаге

Все вершины посещены и обработаны. Алгоритм завершает работу. Как видим, алгоритм успешно нашел кратчайшие пути до других вершин от нашей стартовой вершины с номером 1.

**2 Реализация алгоритмов поиска в глубину и в ширину**

**2.1 Реализация алгоритма DFS для обхода графа на языке С++**

Реализация алгоритма DFS для обхода графа на языке С++ с помощью рекурсии (Приложение А).

Результат выполнения (Рисунок 2.1).

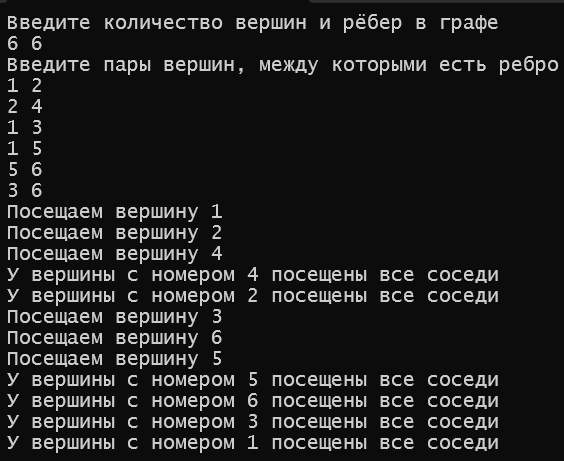


Рисунок 2.1– Результат применения алгоритма DFS с помощью рекурсии

Реализация алгоритма DFS для обхода графа на языке С++, используя стек (Приложение Б).

Результат выполнения (Рисунок 2.2).

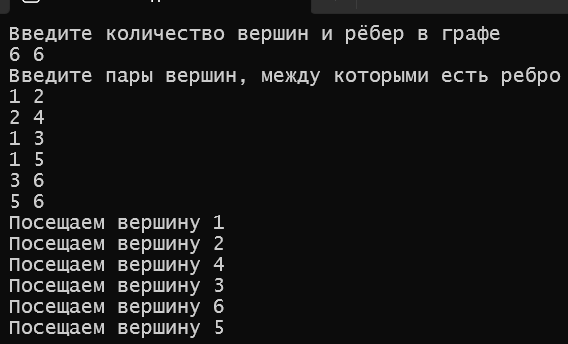


Рисунок 2.2– Результат применения алгоритма DFS, используя стек

Реализация поиска пути из одной вершины в другую, используя DFS на языке С++ (Приложение В).

Результат выполнения (Рисунок 2.3).

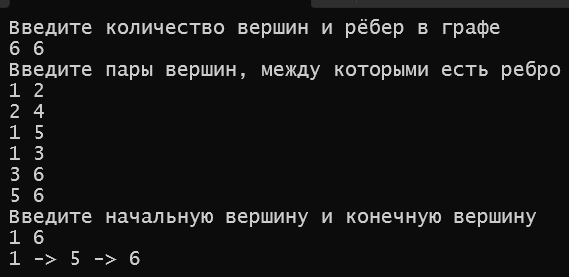


Рисунок 2.3 – Результат поиска пути из одной вершины в другую, используя DFS

Алгоритм поиска в глубину имеет большое практическое значение. Он используется в следующих ситуациях:

1) Для поиска любого маршрута в лабиринте. С помощью DFS мы можем найти случайный путь. Правило прохождения лабиринта в реальной жизни «Идти с левой рукой на стене и всегда поворачивать влево» – пример DFS вне программирования;

2) Для решения задач, связанных с построением маршрута: в сети, на карте, в сервисах покупки билетов и так далее;

3) Как составная часть расчетов в более сложных алгоритмах, например для определения максимального транспортного потока;

4) Для решения ряда задач из теории графов, которые используются в программировании и математике: поиска циклов, сортировки и так далее.

**2.2 Реализация алгоритма BFS для обхода графа на языке С++**

Реализация алгоритма BFS для поиска кратчайших путей на языке С++, используя очередь (Приложение Г).

Результат выполнения (Рисунок 2.4).

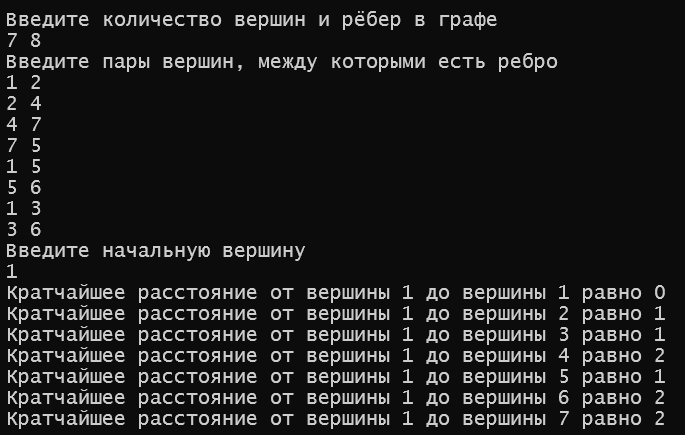


Рисунок 2.4 – Результат применения алгоритма BFS для поиска кратчайших путей, используя очередь

Алгоритм поиска в ширину также имеет большое практическое значение. Его используют:

1) Дата-сайентисты, которые работают с информацией и ее распространением;

2) Разработчики, имеющие дело с определенными видами задач: поиск оптимального маршрута, программирование передвижения «умных» машин, разработка интеллектуальных систем и другие;

3) Математики и другие ученые, которые работают с теорией графов как с фундаментальным научным знанием или в контексте решения практических задач;

4) Инженеры-электроники: конкретно алгоритм BFS используется при трассировке печатных плат;

5) Технические специалисты, работающие в телекоммуникационных системах;

6) Сетевые инженеры, так как теория графов активно используется в сетевых технологиях. BFS, например, применяют при обходе P2P-сетей, или пиринговых сетей, а на них основаны многие сетевые протоколы.

**2.3 Реализация алгоритма Дейкстры для поиска кратчайших путей на языке С++**

Реализация алгоритма Дейкстры для поиска кратчайших путей на языке С++ (Приложение Д).

Результат выполнения (Рисунок 2.5).

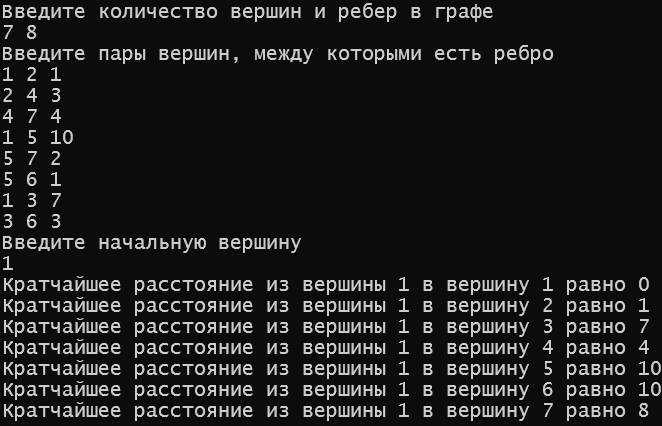


Рисунок 2.5 – Результат применения алгоритма Дейкстры для поиска кратчайших путей

Алгоритм Дейкстры имеет большое практическое значение. Его используют:

1) Математики и другие ученые, которые пользуются графами как абстрактными единицами. Задача поиска маршрута в науке может быть и чисто фундаментальной, и прикладной;

2) Дата-сайентисты. В этой области много математики, в том числе активно используется теория графов;

3) Сетевые инженеры, так как алгоритм Дейкстры лежит в основе работы нескольких протоколов маршрутизации. Сама по себе компьютерная сеть представляет собой граф, поэтому специалисты по сетям должны знать, что это такое.

**Заключение**

В данной работе изучили суть двух очень полезных алгоритмов на графах: поиск в глубину и поиск в ширину, научились реализовывать их на языке C++, узнали, для решения каких задач могут применяться данные алгоритмы.

ПРИЛОЖЕНИЕ А

**Реализация алгоритма DFS для обхода графа на языке С++ с помощью рекурсии**

#include <iostream>

#include <iomanip>

#include <Windows.h>

#include <stdio.h>

#include <vector>

using namespace std;

void DFS(vector<vector<int>>& graph, int vershina, vector<int> &visited) {

cout << "Посещаем вершину " << vershina + 1 << endl;

visited[vershina] = 1;

for (int neighbour : graph[vershina]) {

if(!visited[neighbour]){

DFS(graph, neighbour, visited);

}

}

cout << "У вершины с номером " << vershina + 1 << " посещены все соседи" << endl;

}

int main() {

SetConsoleCP(1251);

SetConsoleOutputCP(1251);

int n, m;

cout << "Введите количество вершин и ребер в графе " << endl;

cin >> n >> m;

vector<int> visited(n);

vector<vector<int>> graph(n);

int a, b;

cout << "Введите пары вершин, между которыми есть ребро" << endl;

for (int j = 0; j < m; j++) {

cin >> a >> b;

a--;

b--;

graph[a].push\_back(b);

graph[b].push\_back(a);

}

for (int i = 0; i < n; i++) {

if (!visited[i]) {

DFS(graph, i, visited);

}

}

}

ПРИЛОЖЕНИЕ Б

**Реализация алгоритма DFS для обхода графа на языке С++, используя стек**

#include <iostream>

#include <iomanip>

#include <Windows.h>

#include <stdio.h>

#include <conio.h>

#include <vector>

#include <stack>

using namespace std;

void DFS(vector<vector<int>>& graph, int vershina, vector<int> &visited) {

stack <int> stack;

stack.push(vershina);

while (!stack.empty()) {

vershina = stack.top();

stack.pop();

if (visited[vershina]) {

continue;

}

visited[vershina] = 1;

cout << "Посещаем вершину " << vershina + 1 << endl;

for (auto t = graph[vershina].rbegin(); t != graph[vershina].rend(); t++) {

if (!visited[\*t]) {

stack.push(\*t);

}

}

}

}

int main() {

SetConsoleOutputCP(1251);

SetConsoleCP(1251);

int n, m;

cout << "Введите количество вершин и ребер в графе" << endl;

cin >> n >> m;

vector <vector <int>> graph(n);

int a, b;

vector <int> visited(n);

cout << "Введите пары вершин, между которыми есть ребро" << endl;

for (int i = 0; i < m; i++) {

cin >> a >> b;

a--; b--;

graph[a].push\_back(b);

graph[b].push\_back(a);

}

for (int i = 0; i < n; i++) {

if (!visited[i]) {

DFS(graph, i, visited);

}

}

}

ПРИЛОЖЕНИЕ В

**Реализация поиска пути из одной вершины в другую, используя DFS на языке С++**

#include <iostream>

#include <iomanip>

#include <Windows.h>

#include <stdio.h>

#include <conio.h>

#include <vector>

using namespace std;

int DFS(vector<vector<int>>& graph, int graph\_size, int start, int end, vector<int> &visited, vector<int> &path) {

if (visited[start]) {

return -1;

}

visited[start] = 1;

if (start == end) {

path[0] = end;

return 0;

}

for (int i = 0; i < size(graph[start]); i++) {

int size\_path = DFS(graph, graph\_size, graph[start][i], end, visited, path);

if (size\_path < 0) {

continue;

}

path[size\_path + 1] = start;

return size\_path + 1;

}

return -1;

}

int main() {

SetConsoleOutputCP(1251);

SetConsoleCP(1251);

int n, m;

cout << "Введите количество вершин и ребер в графе" << endl;

cin >> n >> m;

vector <vector <int>> graph(n);

int a, b;

vector <int> visited(n);

cout << "Введите пары вершин, между которыми есть ребро" << endl;

for (int i = 0; i < m; i++) {

cin >> a >> b;

a--; b--;

graph[a].push\_back(b);

graph[b].push\_back(a);

}

cout << "Введите начальную вершину и конечную вершину" << endl;

cin >> a >> b;

a--; b--;

vector<int> path(n);

int size\_path = DFS(graph, n, a, b, visited, path);

if (size\_path < 0) {

cout << "Пути не существует" << endl;

}

else {

for (int i = size\_path; i >= 0; i--) {

cout << path[i] + 1;

if (i != 0) {

cout << " -> ";

}

else {

cout << endl;

}

}

}

}

ПРИЛОЖЕНИЕ Г

**Реализация алгоритма BFS для поиска кратчайших путей на языке С++, используя очередь**

#include <iostream>

#include <iomanip>

#include <Windows.h>

#include <stdio.h>

#include <conio.h>

#include <vector>

#include <queue>

using namespace std;

const int INF = 1e9;

vector<int> BFS(vector<vector<int>>& graph, int start, vector<int> &dist) {

queue <int> q;

q.push(start);

dist[start] = 0;

while (!q.empty()) {

start = q.front();

q.pop();

for (int neighbour : graph[start]) {

if (dist[neighbour] > dist[start] + 1) {

dist[neighbour] = dist[start] + 1;

q.push(neighbour);

}

}

}

return dist;

}

int main() {

SetConsoleOutputCP(1251);

SetConsoleCP(1251);

int n, m;

cout << "Введите количество вершин и ребер в графе" << endl;

cin >> n >> m;

vector <vector <int>> graph(n);

int a, b;

cout << "Введите пары вершин, между которыми есть ребро" << endl;

for (int i = 0; i < m; i++) {

cin >> a >> b;

a--; b--;

graph[a].push\_back(b);

graph[b].push\_back(a);

}

cout << "Введите начальную вершину" << endl;

cin >> a;

a--;

vector<int> dist(n, INF);

dist = BFS(graph, a, dist);

for (int i = 0; i < n; i++) {

cout << "Кратчайшее расстояние от вершины " << a + 1 << " до вершины " << i + 1 << " равно " << dist[i] << endl;

}

}

ПРИЛОЖЕНИЕ Д

**Реализация алгоритма Дейкстры для поиска кратчайших путей на языке С++**

#include <iostream>

#include<Windows.h>

#include <conio.h>

#include <vector>

using namespace std;

const int INF = 1e9;

vector<int> dejkstra(vector<vector<pair<int, int>>> &graph, int &start, int &n) {

vector<int> dist(n, INF);

dist[start] = 0;

vector<bool> visited(n);

for (int i = 0; i < n; i++) {

int near\_vertex = -1;

for (int vertex = 0; vertex < n; vertex++) {

if (!visited[vertex] and (near\_vertex == -1 or dist[near\_vertex] > dist[vertex])) {

near\_vertex = vertex;

}

}

visited[near\_vertex] = true;

for (pair<int,int> to : graph[near\_vertex]) {

if (dist[to.first] > dist[near\_vertex] + to.second) {

dist[to.first] = dist[near\_vertex] + to.second;

}

}

}

return dist;

}

int main()

{

SetConsoleCP(1251);

SetConsoleOutputCP(1251);

int n, m;

cout << "Введите количество вершин и ребер в графе\n";

cin >> n >> m;

vector<vector<pair<int, int>>> graph(n);

cout << "Введите пары вершин, между которыми есть ребро\n";

for (int i = 0; i < m; i++) {

int a, b, weight;

cin >> a >> b >> weight;

a--;

b--;

graph[a].push\_back({ b, weight });

graph[b].push\_back({ a, weight });

}

int start;

cout << "Введите начальную вершину\n";

cin >> start;

start--;

vector<int> dist = dejkstra(graph, start, n);

for (int i = 0; i < n; i++) {

if (dist[i] != INF) {

cout << "Кратчайшее расстояние из вершины " << start + 1<< " в вершину " << i + 1 << " равно " << dist[i] << '\n';

}

else {

cout << "Пути из вершины " << start + 1 << " в вершину " << i + 1 << " не существует\n";

}

}

}