МЕЖГОСУДАРСТВЕННОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ
ВЫСШЕГО ОБРАЗОВАНИЯ
«БЕЛОРУССКО-РОССИЙСКИЙ УНИВЕРСИТЕТ»

Кафедра «Автоматизированные системы управления»

ТЕОРИЯ ГРАФОВ

Методические рекомендации к лабораторным работам для студентов специальности 1-53 01 02 «Автоматизированные системы обработки информации» очной и заочной форм обучения

Могилев 2023

УДК 519.17
ББК 22.176
Т33

Рекомендовано к изданию
учебно-методическим отделом
Белорусско-Российского университета

Одобрено кафедрой «Автоматизированные системы управления» «20» декабря 2022 г., протокол № 5

Составитель д-р техн. наук, доц. А. И. Якимов

Рецензент канд. техн. наук, доц. С. К. Крутолевич

В методических рекомендациях к лабораторным работам по дисциплине «Теория графов» (3 семестр) приведены задания и контрольные вопросы для самостоятельной подготовки.

Учебно-методическое издание

ТЕОРИЯ ГРАФОВ

Ответственный за выпуск А. И. Якимов

Корректор А. А. Подошевко

Компьютерная верстка Н. П. Полевничая

Подписано в печать . Формат 60×84/16. Бумага офсетная. Гарнитура Таймс.
Печать трафаретная. Усл. печ. л. . Уч.-изд. л. . Тираж 21 экз. Заказ №

Издатель и полиграфическое исполнение:
Межгосударственное образовательное учреждение высшего образования
«Белорусско-Российский университет».
Свидетельство о государственной регистрации издателя,
изготовителя, распространителя печатных изданий
№ 1/156 от 07.03.2019.
Пр-т Мира, 43, 212022, г. Могилев.

© Белорусско-Российский
университет, 2023

Содержание

Введение 4
1 Лабораторная работа № 1. Основы языка Python и библиотеки для работы с графами 5
2 Лабораторная работа № 2. Операции над графами 12
3 Лабораторная работа № 3. Представление и реализация деревьев 18
4 Лабораторная работа № 4. Элементарные алгоритмы для работы с графами 25
5 Лабораторная работа № 5. Оптимизационные алгоритмы для работы с графами. Кратчайшие пути в графе.. 35
6 Лабораторная работа № 6. Оптимизационные алгоритмы для работы с графами. Остовные деревья минимальной стоимости..................... 41
7 Лабораторная работа № 7. Поиск компонент сильной связности графа 43
8 Лабораторная работа № 8. Задача о максимальном потоке....................... 45
Список литературы 47

Введение

Целью преподавания дисциплины «Теория графов» является обучение студентов основным методам теории графов для решения задач из области программирования, администрирования сетей, информационных потоков, планирования, проектирования и управления автоматизированными системами.

Цель методических рекомендаций – помочь студентам в самостоятельной подготовке и выполнении задания к лабораторным занятиям по дисциплине.

Порядок выполнения каждой лабораторной работы.

1 Изучить теоретические сведения.

2 Получить задание у преподавателя, выполнить в соответствии с заданным вариантом.

3 Сделать выводы по результатам выполнения задания.

4 Оформить отчет.

Содержание отчета.

1 Цель работы.

2 Постановка задачи.

3 Результаты выполнения задания.

4 Выводы.

1 Лабораторная работа № 1. Основы языка Python и библиотеки для работы с графами

Цель работы: получить базовое представление о языке программирования Python, а также научиться применять его в качестве инструмента для работы с графами.

Основные теоретические положения

Python не требует явного объявления переменных, является регистро-зависимым (переменная var не эквивалентна переменной Var или VAR – это три разные переменные) *объектно-ориентированным* языком.

Синтаксис.

Python не содержит операторных скобок (begin..end в Pascal или {..} в Си), вместо этого блоки выделяются отступами: пробелами или табуляцией, а вход в блок из операторов осуществляется двоеточием. Однострочные комментарии начинаются со знака фунта «#», многострочные начинаются и заканчиваются тремя двойными кавычками «"""». Чтобы присвоить значение переменной используется знак «=», а для сравнения – «==». Для увеличения значения переменной или добавления к строке используется оператор «+=», а для уменьшения – «–=». Все эти операции могут взаимодействовать с большинством типов, в том числе со строками. Например:

```
>>> myvar = 3
>>> myvar += 2
>>> myvar –= 1
""" «Это многострочный комментарий
Строки, заключенные в три двойные кавычки игнорируются»"""
>>> mystring = «Hello»
>>> mystring += " world."
>>> print mystring
Hello world.
# Следующая строка меняет
значения переменных местами. (Всего одна строка!)
>>> myvar, mystring = mystring, myvar
```

Типы данных и структуры данных.

Переменная хранит данные одного из типов данных. В Python существует множество различных типов данных. Самые базовые типы: bool, int, float, complex и str.

Python содержит такие структуры данных, как списки (lists), кортежи (tuples) и словари (dictionaries). Списки похожи на одномерные массивы (но можно использовать Список, включающий списки, – многомерный массив), кортежи – неизменяемые списки, словари – тоже списки, но индексы могут быть любого типа, а не только числовыми. «Массивы» в Python могут содержать данные любого типа, т. е. в одном массиве могут находиться числовые, строковые и другие типы данных. Массивы начинаются с индекса 0, а последний элемент

можно получить по индексу –1. Можно присваивать переменным функции и применять их соответственно.

Можно использовать часть массива, задавая первый и последний индексы через двоеточие «:». В таком случае получают часть массива, от первого индекса до второго не включительно. Если не указан первый элемент, то отсчет начинается с начала массива, а если не указан последний – то массив считывается до последнего элемента. Отрицательные значения определяют положение элемента с конца. Например:

```
>>> mylist = [«List item 1», 2, 3.14]
>>> print mylist[:] #Считываются все элементы массива
['List item 1', 2, 3.1400000000000001]
>>> print mylist[0:2] #Считываются нулевой и первый элементы массива.
['List item 1', 2]
>>> print mylist[–3:–1] #Считываются элементы от нулевого (–3) до второго (–1) (не включительно)
['List item 1', 2]
>>> print mylist[1:] #Считываются элементы от первого до последнего
[2, 3.14]
```

Строки.

Строки в Python обособляются кавычками двойными «"» или одинарными «'». Внутри двойных кавычек могут присутствовать одинарные или наоборот. Например, строка «Он сказал 'Привет!'» будет выведена на экран как «Он сказал 'Привет!'». Если нужно использовать строку из несколько строк, то эту строку надо начинать и заканчивать тремя двойными кавычками «"""». Можно подставить в шаблон строки-элементы из кортежа или словаря. Знак процента «%» между строкой и кортежем заменяет в строке символы «%s» на элемент кортежа. Словари позволяют вставлять в строку элемент под заданным индексом. Для этого надо использовать в строке конструкцию «%(индекс)s». В этом случае вместо «%(индекс)s» будет подставлено значение словаря под заданным индексом.

```
>>>print «Name: %s\nNumber: %s\nString: %s» % (myclass.name, 3, 3 * "–")
Name: Poromenos
Number: 3
String: —
strString = ""«Этот текст расположен
на нескольких строках»""
>>> print «This %(verb)s a %(noun)s.» % {«noun»: «test», «verb»: «is»}
This is a test.
```

Модули.

Модуль math – один из наиважнейших в Python. Этот модуль предоставляет обширный функционал для работы с числами.

math.ceil(X) – округление до ближайшего большего числа.

math.copysign(X, Y) – возвращает число, имеющее модуль такой же, как и у числа X, а знак – как у числа Y.

math.fabs(X) – модуль X.

math.factorial(X) – факториал числа X.

math.floor(X) – округление вниз.

math.fmod(X, Y) – остаток от деления X на Y.

math.frexp(X) – возвращает мантиссу и экспоненту числа.

math.ldexp(X, I) – X * 2i. Функция, обратная функции math.frexp().

math.fsum(последовательность) – сумма всех членов последовательности. Эквивалент встроенной функции sum(), но math.fsum() более точна для чисел с плавающей точкой.

math.isfinite(X) – является ли X числом.

math.isinf(X) – является ли X бесконечностью.

math.isnan(X) – является ли X NaN (Not a Number – не число).

math.modf(X) – возвращает дробную и целую часть числа X. Оба числа имеют тот же знак, что и X.

math.trunc(X) – усекает значение X до целого.

math.exp(X) – eX.

math.expm1(X) – eX – 1. При X → 0 точнее, чем math.exp(X)–1.

math.log(X, [base]) – логарифм X по основанию base. Если base не указан, вычисляется натуральный логарифм.

math.log1p(X) – натуральный логарифм (1 + X). При X → 0 точнее, чем math.log(1+X).

math.log10(X) – логарифм X по основанию 10.

math.log2(X) – логарифм X по основанию 2.

math.pow(X, Y) – XY.

math.sqrt(X) – квадратный корень из X.

math.acos(X) – арккосинус X. В радианах.

math.asin(X) – арксинус X. В радианах.

math.atan(X) – арктангенс X. В радианах.

math.atan2(Y, X) – арктангенс Y/X. В радианах. С учетом четверти, в которой находится точка (X, Y).

math.cos(X) – косинус X (X указывается в радианах).

math.sin(X) – синус X (X указывается в радианах).

math.tan(X) – тангенс X (X указывается в радианах).

math.hypot(X, Y) – вычисляет гипотенузу треугольника с катетами X и Y (math.sqrt(x * x + y * y)).

math.degrees(X) – конвертирует радианы в градусы.

math.radians(X) – конвертирует градусы в радианы.

math.cosh(X) – вычисляет гиперболический косинус.

math.sinh(X) – вычисляет гиперболический синус.

math.tanh(X) – вычисляет гиперболический тангенс.

math.acosh(X) – вычисляет обратный гиперболический косинус.

math.asinh(X) – вычисляет обратный гиперболический синус.

math.atanh(X) – вычисляет обратный гиперболический тангенс.

math.erf(X) – функция ошибок.

math.erfc(X) – дополнительная функция ошибок (1 – math.erf(X)).

math.gamma(X) – гамма-функция X.

math.lgamma(X) – натуральный логарифм гамма-функции X.

math.pi – pi = 3,1415926...
math.e – e = 2,718281...

Операторы.

If:

Условные конструкции используют условные выражения и в зависимости от их значения направляют выполнение программы по одному из путей. Одна из таких конструкций – это конструкция if. Она имеет следующее формальное определение:

```
if логическое_выражение:
    инструкции
[elif логическое выражение:
    инструкции]
[else:
    инструкции]
```

В самом простом виде после ключевого слова if идет логическое выражение. И если это логическое выражение возвращает True, то выполняется последующий блок инструкций, каждая из которых должна начинаться с новой строки и должна иметь отступы от начала выражения if (отступ желательно делать в четыре пробела или то количество пробелов, которое кратно 4).

While:

Цикл while проверяет истинность некоторого условия, и если условие истинно, то выполняет инструкции цикла. Он имеет следующее формальное определение:

```
while условное_выражение:
    инструкции
```

После ключевого слова while указывается условное выражение, и пока это выражение возвращает значение True, будет выполняться блок инструкций, который идет далее.

Все инструкции, которые относятся к циклу while, располагаются на последующих строках и должны иметь отступ от начала ключевого слова while.

For:

Другой тип циклов представляет конструкция for. Этот цикл пробегается по набору значений, помещает каждое значение в переменную, и затем в цикле мы можем с этой переменной производить различные действия. Формальное определение цикла for:

```
for переменная in набор_значений:
    инструкции
```

После ключевого слова for идет название переменной, в которую будут помещаться значения. Затем после оператора in указывается набор значений и двоеточие. А со следующей строки располагается блок инструкций цикла, которые также должны иметь отступы от начала цикла.

При выполнении цикла Python последовательно получает все значения из набора и передает их переменную. Когда все значения из набора будут перебраны, цикл завершает свою работу.

Функции.

Функции представляют блок кода, который выполняет определенную задачу и который можно повторно использовать в других частях программы. В предыдущих статьях уже использовались функции. В частности, функция print(), которая выводит некоторое значение на консоль. Python имеет множество встроенных функций и позволяет определять свои функции. Формальное определение функции:

```
def имя_функции ([параметры]):
    инструкции
```

Определение функции начинается с выражения def, которое состоит из имени функции, набора скобок с параметрами и двоеточия. Параметры в скобках необязательны. А со следующей строки идет блок инструкций, которые выполняет функция. Все инструкции функции имеют отступы от начала строки.

Классы.

Язык Python ограничен в множественном наследовании в классах. Внутренние переменные и внутренние методы классов начинаются с двух знаков нижнего подчеркивания «__» (например, «__myprivatevar»). Можно также присвоить значение переменной класса извне.

Подключение библиотек.

Подключить модуль можно с помощью инструкции import. После ключевого слова **import** указывается название модуля. Одной инструкцией можно подключить несколько модулей, хотя этого не рекомендуется делать, т. к. это снижает читаемость кода.

```
>>> import time, random
```

Библиотеки для работы с графами.

Для работы с графами в языке программирования Python мы предлагаем использовать библиотеки NetworkX, Matplotlib и graphviz.

NetworkX – библиотека, специально предназначена для работы с графами. graphviz и Matplotlib – библиотеки, предназначенные для визуализации графа.

NetworkX.

Библиотека networkX создана на языке Python и предназначена для работы с графами и другими сетевыми структурами. Это свободное ПО, распространяемое под новой BSD-лицензией.

Основные возможности библиотеки.

1 Классы для работы с простыми, ориентированными и взвешенными графами.

2 Узлом может быть практически что угодно: time-series, текст, изображение, XML.

3 Сохранение / загрузка графов в/из наиболее распространённых форматов файлов хранения графов.

4 Встроенные процедуры для создания графов базовых типов.

5 Методы для обнаружения подграфов, клик и K-дольных графов (K-core) (максимальный подграф, в котором каждая вершина имеет по крайней мере уровень K).

6 Получение таких характеристик графа, как степени вершин, высота графа, диаметр, радиус, длины путей, центр, промежуточности и т. д.

7 Визуализировать сети в виде 2D- и 3D-графиков.

И многое другое … .

Matplotlib.

Библиотека matplotlib – это библиотека двумерной графики для языка программирования Python, с помощью которой можно создавать высококачественные рисунки различных форматов. Matplotlib представляет собой модуль-пакет для Python.

Matplotlib состоит из множества модулей. Модули наполнены различными классами и функциями, которые иерархически связаны между собой.

Создание рисунка в matplotlib схоже с рисованием в реальной жизни. Так, художнику нужно взять основу (холст или бумагу), инструменты (кисти или карандаши), иметь представление о будущем рисунке (что именно он будет рисовать) и, наконец, выполнить всё это и нарисовать рисунок деталь за деталью.

В matplotlib все эти этапы также существуют, и в качестве художника-исполнителя здесь выступает сама библиотека. От пользователя требуется управлять действиями художника-matplotlib, определяя что именно он должен нарисовать и какими инструментами. Обычно создание основы и процесс непосредственно отображения рисунка отдаёт полностью на откуп matplotlib. Таким образом, пользователь библиотеки matplotlib выступает в роли управленца. И чем проще ему управлять конечным результатом работы matplotlib (рисунок 1), тем лучше.

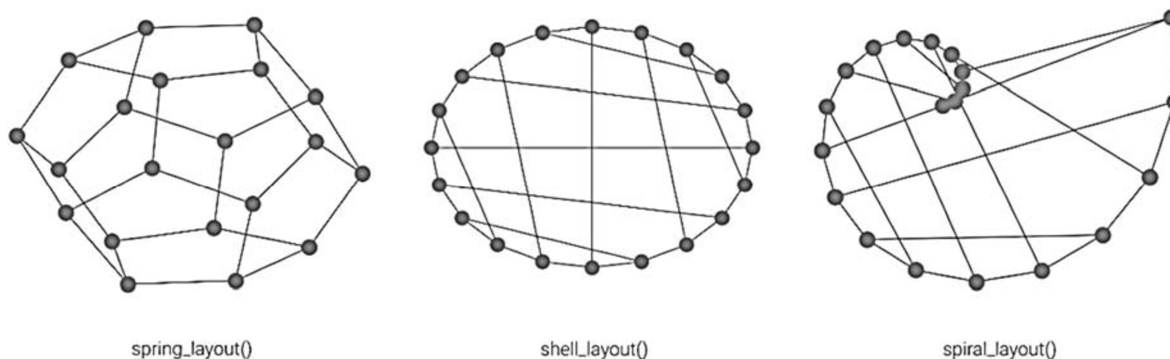


Рисунок 1 – Пример результата работы matplotlib

Graphviz.

В последнее время при визуализации данных используется библиотека graphviz, которая используется для создания деревьев решений и эффектов блок-схем. Движок диаграмм использует язык описания графов DOT, который представляет собой текстовое описание структуры графа: вершины, их связи, группы и атрибуты для их визуального оформления.

Практические задания

Задание 1

Написать программу, которая запрашивает исходные данные и получает вычисленный результат.

1 Написать программу пересчета веса из фунтов в килограммы (1 фунт = 0,4536 кг).

2 Написать программу пересчета расстояния из милей в километры (1 миля = 1,609 км).

3 Перевести дозу радиоактивного излучения из микрозивертов в миллирентгены (1 мкЗв = 0,115 мР).

4 Перевести температуру из градусов Кельвина в градусы Цельсия (0 ºС = –273,1 К).

5 Написать программу пересчета объема из галлонов в литры (1 галлон = 3,785 л).

6 Написать программу пересчета расстояния из морских лиг в километры (1 морская лига = 5,556 км).

7 Написать программу пересчета веса из унций в граммы (1 унция = 28,35 г).

8 Написать программу пересчета расстояния из кабельтов в метры (1 кабельтов = 219,5 м).

9 Написать программу пересчета расстояния из морских миль в километры (1 морская миля = 1,852 км).

10 Написать программу пересчета длины из ярдов в метры (1 ярд = 0,9144 м).

Задание 2

1 Прямоугольный треугольник задан гипотенузой и одним из катетов. Найти второй катет и площадь треугольника.

2 Треугольник задан величинами своих углов (в градусах) и радиусом описанной окружности. Найти длины сторон треугольника.

3 Прямоугольный треугольник задан двумя катетами. Вычислить радиусы вписанной и описанной окружностей.

4 Вычислить длину окружности, площадь круга и объем шара одного и того же заданного радиуса.

5 Вычислить периметр и площадь прямоугольного треугольника по заданным катетам треугольника.

6 Вычислить периметр и площадь треугольника по заданным координатам трех его вершин.

7 Треугольник задан длинами своих сторон. Найти длины его высот.

8 Треугольник задан длинами своих сторон. Найти длины его медиан.

9 Треугольник задан длинами своих сторон. Найти длины его биссектрис.

10 Треугольник задан длинами своих сторон. Найти радиусы описанной и вписанной окружностей треугольника.

Контрольные вопросы

1 Какие структуры данных содержит Python?
2 Как обособляются строки в Python?
3 Какие условные конструкции и циклы есть в Python?
4 Как подключить библиотеки в Python?
5 Какие библиотеки могут помочь в работе с графами?
6 Какие операции над графами позволяют делать библиотеки?

2 Лабораторная работа № 2. Операции над графами

Цель работы: изучить операции над графами для образования новых графов из нескольких более простых.

Основные теоретические положения

Удаление вершины. Пусть $G\ (V, E)$ – граф и $a \in V$. Удалить вершину a из графа G – это значит построить новый граф $G_1(V_1, E_1)$, в котором $V_1 = V(G)\setminus\{a\}$ и E_1 получается из E удалением всех ребер, инцидентных вершине a (рисунок 2).

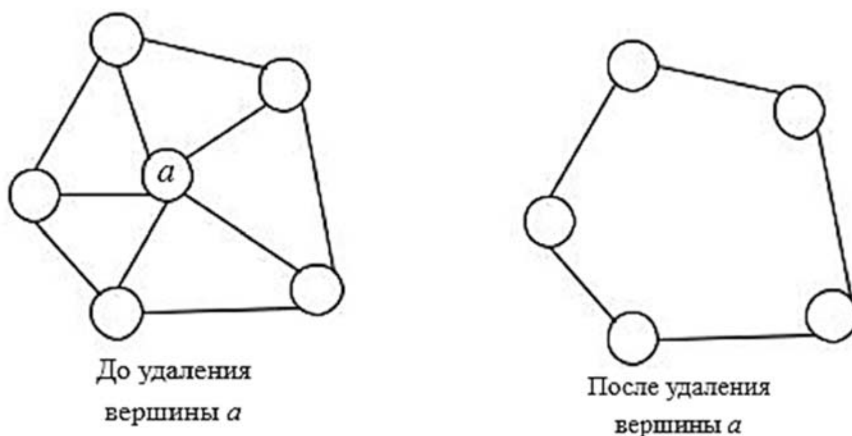


Рисунок 2 – Пример удаления вершины

Удаление ребра. Пусть $G\ (V, E)$ – граф и $b \in E$. Удалить ребро b – это значит построить новый граф $G_1(V_1, E_1)$, в котором $V_1 = V$ и $E_1 = E \setminus\{b\}$ (рисунок 3).

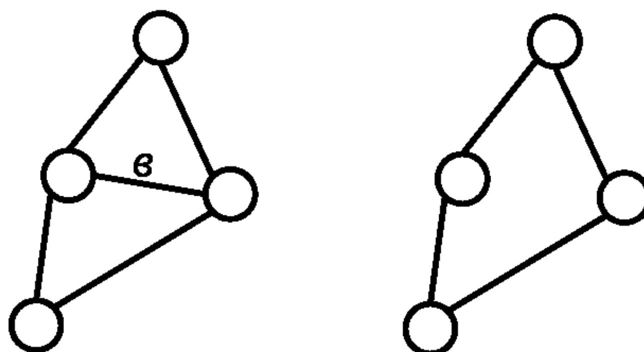


Рисунок 3 – Пример удаления ребра

Дополнением графа G называется граф G_1 с теми же вершинами, что и граф G, и с теми и только теми ребрами, которые необходимо добавить к графу G, чтобы получился полный граф (рисунок 4).

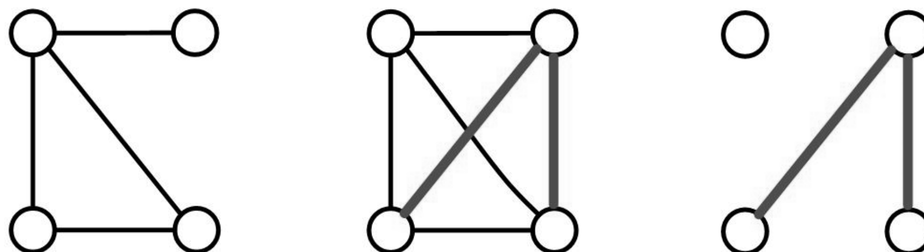


Рисунок 4 – Пример операции дополнения графа

Например, дополнением полного графа на n вершинах будет пустой граф на n вершинах и наоборот.

Пусть $G_1(X_1, E_1)$ и $G_2(X_2, E_2)$ – произвольные графы. Объединением $G_1 \cup G_2$ графов G_1 и G_2 называется граф с множеством вершин $X_1 \cup X_2$ и с множеством ребер (дуг) $E_1 \cup E_2$.

Рассмотрим операцию на примере графов $G_1(X_1, E_1)$ и $G_2(X_2, E_2)$, приведенных на рисунке 5. Множества вершин первого и второго графов определяются как $X_1 = \{x_1, x_2, x_3\}$ и $X_2 = \{x_2, x_3, x_4\}$, а множество вершин результирующего графа – $X = X_1 \cup X_2 = \{x_1, x_2, x_3, x_4\}$. Аналогично определяем множества дуг графа:

$E_1 = \{(x_1, x_2), (x_1, x_3), (x_2, x_1), (x_3, x_3)\}$; $E_2 = \{(x_2, x_4), (x_3, x_2), (x_4, x_3)\}$;

$$E = \{(x_1, x_2), (x_1, x_3), (x_2, x_1), (x_3, x_3), (x_2, x_4), (x_3, x_2), (x_4, x_3)\}.$$

Результирующий граф $G(X, E) = G_1(X_1, E_1) \cup G_2(X_2, E_2)$ также приведен на рисунке 5.

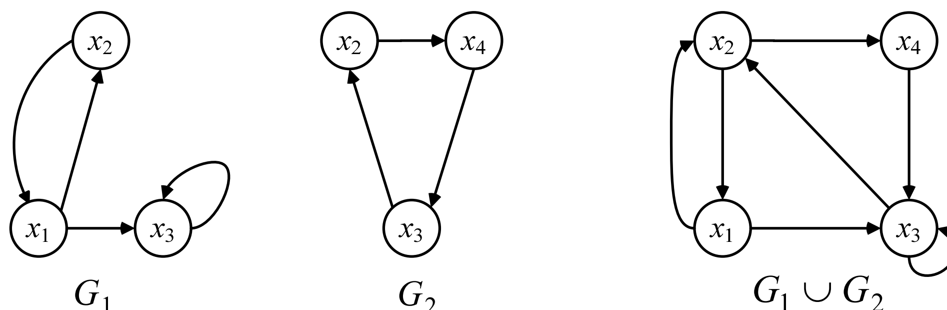


Рисунок 5 – Операция объединения графов

Операция объединения обладает следующими свойствами, которые следуют из определения операции и свойств операций на множествах:

$G_1 \cup G_2 = G_2 \cup G_1$ – свойство коммутативности;

$G_1 \cup (G_2 \cup G_3) = (G_1 \cup G_2) \cup G_3$ – свойство ассоциативности.

Операция объединения графов может быть выполнена в матричной форме.

Пусть $G_1(X_1, E_1)$ и $G_2(X_2, E_2)$ – произвольные графы. Пересечением $G_1 \cap G_2$ графов G_1 и G_2 называется граф с множеством вершин $X_1 \cap X_2$, с множеством ребер (дуг) $E = E_1 \cap E_2$.

Операция пересечения обладает свойствами, которые следуют из определения операции и свойств операций на множествах:

$G_1 \cap G_2 = G_2 \cap G_1$ – свойство коммутативности;

$G_1 \cap (G_2 \cap G_3) = (G_1 \cap G_2) \cap G_3$ – свойство ассоциативности.

Для того чтобы операция пересечения была всеобъемлющей, необходимо ввести понятие пустого графа. Граф $G(X, E)$ называется пустым, если множество X вершин графа является пустым ($X = \varnothing$). Заметим, что в этом случае и множество E ребер (дуг) графа также пустое множество ($E = \varnothing$). Пустой граф обозначается символом $\varnothing$. Такой граф может быть получен в результате выполнения операции пересечения графов, у которых $X_1 \cap X_2=\varnothing$. В этом случае говорят о непересекающихся графах.

Рассмотрим выполнение операции пересечения графов, изображенных на рисунке 6. Для нахождения множества вершин результирующего графа запишем множества вершин исходных графов и выполним над этими множествами операцию пересечения:

$$X_1 = \{x_1, x_2, x_3\};\ X_2 = \{x_1, x_2, x_3, x_4\};$$

$$X = X_1 \cap X_2 = \{x_1, x_2, x_3\}.$$

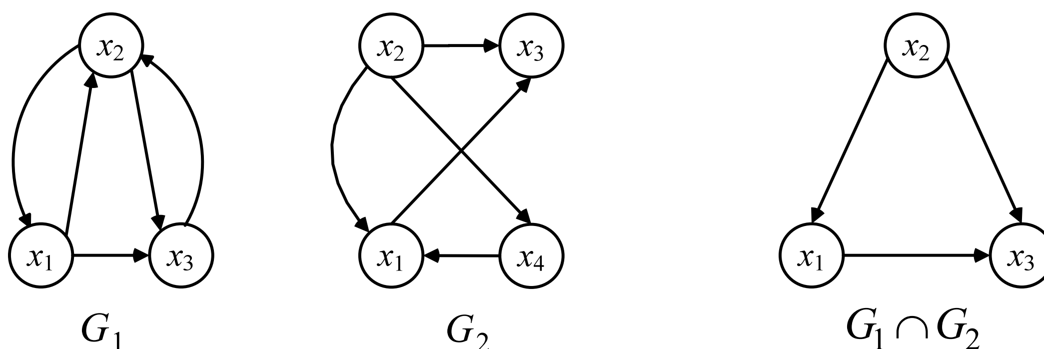


Рисунок 6 – Операция пересечения графов

Аналогично определяем множество E дуг результирующего графа:

$$E_1 = \{(x_1, x_2), (x_1, x_3), (x_2, x_1), (x_2, x_3), (x_3, x_2)\};$$

$$E_2 = \{(x_1, x_3), (x_2, x_1), (x_2, x_3), (x_2, x_4), (x_4, x_1)\};$$

$$E = E_1 \cap E_2 = \{(x_1, x_3), (x_2, x_1)\}.$$

Графы $G_1(X_1, E_1)$, $G_2(X_2, E_2)$ и их пересечение приведены на рисунке 6.

Операция пересечения графов может быть выполнена в матричной форме.

Пусть $G_1(X, E_1)$ и $G_2(X, E_2)$ – два графа с одним и тем же множеством вершин X. Композицией $G_1(G_2)$ графов G_1 и G_2 называется граф с множеством дуг E, в котором существует дуга (x_i, x_j) тогда и только тогда, когда существует дуга (x_i, x_k), принадлежащая множеству E_1, и дуга (x_k, x_j), принадлежащая множеству E_2.

Рассмотрим выполнение операции композиции $G_1(G_2)$ на графах, изображенных на рисунке 7. Для рассмотрения операции составим таблицу 1, в первом столбце которой указываются дуги (x_i, x_k), принадлежащие графу G_1, во втором – дуги (x_k, x_j), принадлежащие графу G_2, а в третьем – результирующее ребро (x_i, x_j) для графа $G_1(G_2)$.

Таблица 1 – Композиция графов

G_1	G_2	$G_1(G_2)$
(x_1, x_2)	(x_2, x_1) (x_2, x_3)	(x_1, x_1) (x_1, x_3)
(x_1, x_3)	(x_3, x_3)	(x_1, x_3)
(x_2, x_1)	(x_1, x_1) (x_1, x_3)	(x_2, x_1) (x_2, x_3)

Заметим, что дуга (x_1, x_3) результирующего графа в таблице встречается дважды. Однако поскольку рассматриваются графы без параллельных ребер (дуг), то в множестве E результирующего графа дуга (x_1, x_3) учитывается только один раз, т. е. $E = \{(x_1, x_1), (x_1, x_3), (x_2, x_1), (x_2, x_3)\}$.

На рисунке 7 изображены графы G_1, G_2 и их композиция $G_1(G_2)$. На этом же рисунке изображен граф $G_2(G_1)$. Рекомендуется самостоятельно построить граф $G_2(G_1)$ и убедиться, что графы $G_1(G_2)$ и $G_2(G_1)$ неизоморфны.

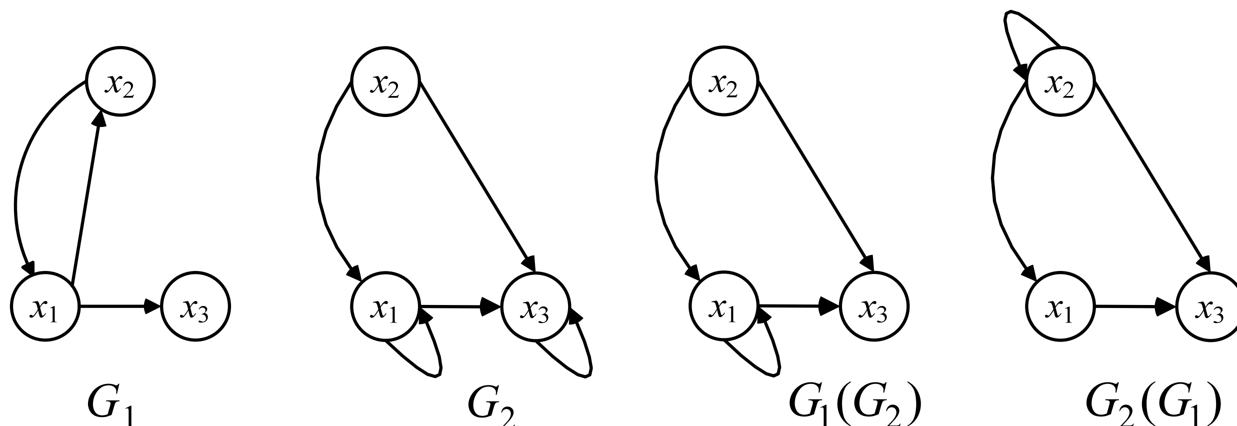


Рисунок 7 – Композиции графов

Пусть $G = (V, E)$ – орграф, где $V = \{v_1, \dots, v_n\}$. Матрицей достижимости орграфа G называется квадратная матрица $T(G) = [t_{ij}]$ порядка n, у которой $t_{ij} = 1$, если вершина v_j достижима из v_i, и $t_{ij} = 0$ – в противном случае.

Матрицей сильной связности орграфа G называется квадратная матрица $S(G) = [s_{ij}]$ порядка n, у которой $s_{ij} = 1$, если вершина v_i достижима из v_j и

одновременно вершина v_j достижима из v_i, и $s_{ij} = 0$ – в противном случае, т. е. $s_{ij} = 1$ тогда и только тогда, когда вершины v_i, v_j принадлежат одной компоненте сильной связности орграфа G.

Пусть $G = (V, E)$ – орграф, где $V = \{v_1, \ldots, v_n\}$. Матрицей связности графа G называется квадратная матрица $S(G) = [s_{ij}]$ порядка n, у которой $s_{ij} = 1$, если $i = j$ или существует маршрут, соединяющий вершины v_i и v_j, $s_{ij} = 0$ – в противном случае (т. е. $s_{ij} = 1$ тогда и только тогда, когда вершины v_i, v_j принадлежат одной компоненте связности орграфа G).

Практические задания

Даны графы G_1 и G_2. Найти $G_1 \cup G_2$, $G_1 \cap G_2$, G_1 (G_2), G_2 (G_1). Для графа $G_1 \cup G_2$ найти матрицы смежности, инцидентности, сильных компонент (рисунок 8).

Вариант	Графы G_1 и G_2	Вариант	Графы G_1 и G_2
1	G_1: 1 2 4 3 G_2: 1 3 2	**6**	G_1: 1 2 4 3 G_2: 1 3 2
2	G_1: 1 2 4 3 G_2: 1 3 2	**7**	G_1: 1 2 4 3 G_2: 1 3 2
3	G_1: 1 2 4 3 G_2: 1 3 2	**8**	G_1: 1 2 4 3 G_2: 1 3 2
4	G_1: 1 2 4 3 G_2: 1 3 2	**9**	G_1: 1 2 4 3 G_2: 1 3 2
5	G_1: 1 2 4 3 G_2: 1 3 2	**10**	G_1: 1 2 4 3 G_2: 1 3 2

Рисунок 8 – Варианты контрольных заданий

Вариант	Графы G_1 и G_2	Вариант	Графы G_1 и G_2
11	G_1: G_2:	**19**	G_1: G_2:
12	G_1: G_2:	**20**	G_1: G_2:
13	G_1: G_2:	**21**	G_1: G_2:
14	G_1: G_2:	**22**	G_1: G_2:
15	G_1: G_2:	**23**	G_1: G_2:
16	G_1: G_2:	**24**	G_1: G_2:
17	G_1: G_2:	**25**	G_1: G_2:
18	G_1: G_2:	**26**	G_1: G_2:

Окончание рисунка 8

3 Лабораторная работа № 3. Представление и реализация деревьев

Цель работы: сформировать знания и умения по представлению и реализации деревьев.

Основные теоретические положения

Дерево представляет собой совокупность узлов (вершин) и связывающих их ребер, которая характеризуется следующими свойствами:

1) существует единственный узел (он называется корнем дерева), на который не ссылается никакой другой узел;

2) начиная с корня и следуя по определенной цепочке указателей, содержащихся в узлах, можно осуществить доступ к любому узлу структуры (связность);

3) на каждый элемент, кроме корня, имеется единственная ссылка, т. е. каждый элемент адресуется единственным указателем (ацикличность).

Дерево называется бинарным (двоичным), если каждый его узел имеет не более двух потомков. Дерево любого вида можно преобразовать единственным образом в эквивалентное бинарное дерево.

Бинарное дерево поиска имеет структуру бинарного дерева, но элементы в нем расположены по определенным правилам.

1 Любое значение меньше значения узла является старшим сыном этого узла или его потомком.

2 Любое значение, больше или равное значению узла, становится младшим сыном этого узла или его потомком. В бинарном дереве поиска все узлы с ключом меньше, чем у корня, находятся в левом поддереве; все узлы с ключом больше, чем у корня, – в правом.

Рассмотрим алгоритмы прохождения всех узлов дерева (обхода), обход помогает вывести дерево. Существуют прямой обход, обратный и симметричный. Определим эти способы рекуррентно: если дерево T нулевое, то в список обхода заносится нулевая запись; если дерево T состоит из одного узла, тогда в список обхода записывается этот узел; далее, пусть T – дерево с корнем n и поддеревьями T_1, T_2, …, T_k (рисунок 9). Тогда имеем следующее:

– при прямом обходе узлов дерева T сначала посещаем корень n, затем – узлы поддерева T_1, далее – все узлы поддерева T_2 и т. д. Последними посещаются узлы поддерева T_k (все – в прямом порядке);

– при обратном обходе узлов дерева T сначала посещаем в обратном порядке все узлы поддерева T_1, затем последовательно посещаем в обратном порядке все узлы поддеревьев T_2, …, T_k, последним посещается корень n;

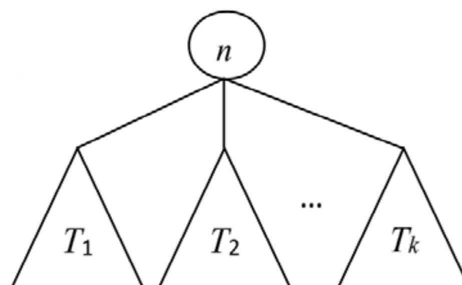


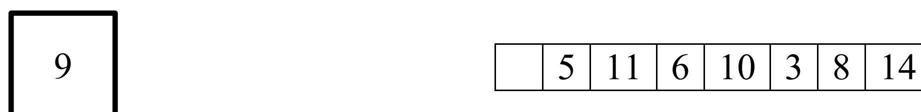
Рисунок 9 – Дерево

– при симметричном обходе узлов дерева T сначала посещаем в симметричном порядке все

узлы поддерева T_1, далее – корень n, затем последовательно посещаем в симметричном порядке все узлы поддеревьев T_2, …, T_k.

Пример 1 – Реализация алгоритма построения бинарного дерева поиска на практике.

Дана таблица значений (рисунок 10), из которых нужно построить бинарное дерево поиска.



9	5	11	6	10	3	8	14

Рисунок 10 – Таблица значений для построения бинарного дерева

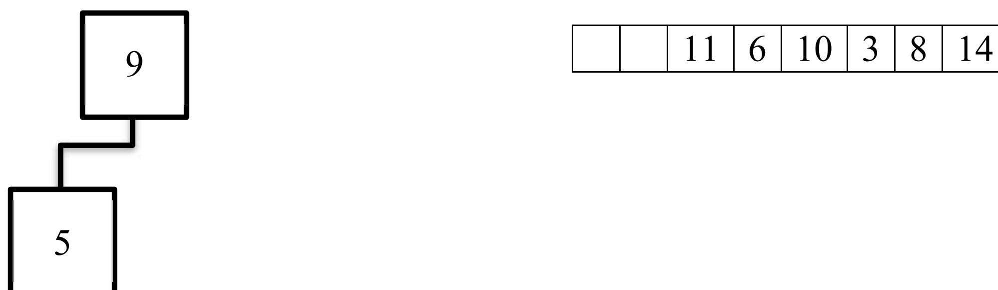
Шаг 1. Начнём строить дерево, первое значение в таблице будет его корнем (рисунок 11).

9

	5	11	6	10	3	8	14

Рисунок 11 – Бинарное дерево на шаге 1

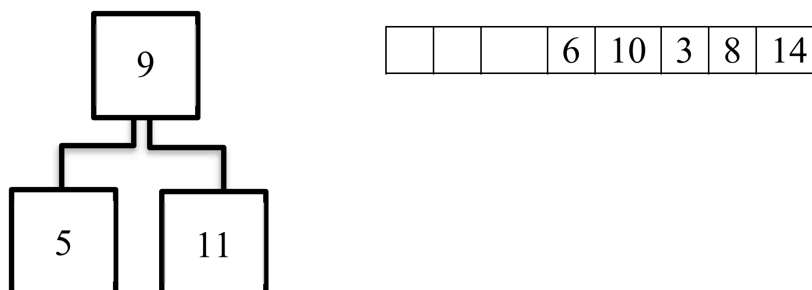
Шаг 2. Берём следующие значение из таблицы и сравниваем его со значением корня, если значение больше корня, то оно становится правым ребенком, если меньше – левым (рисунок 12).



		11	6	10	3	8	14

Рисунок 12 – Бинарное дерево на шаге 2

Шаг 3. Значение 11 > 9, следовательно, оно становится правым ребенком (рисунок 13).



			6	10	3	8	14

Рисунок 13 – Бинарное дерево на шаге 3

Шаг 4. Значение $6 < 9$, следовательно оно должно стать левым ребенком, однако у 9 уже есть левый ребенок. В таком случае мы должны сравнивать 6 с 5. Если значение больше корня, то оно становится правым ребенком, если меньше – левым. В нашем случае 6 становится правым ребёнком 5 (рисунок 14).

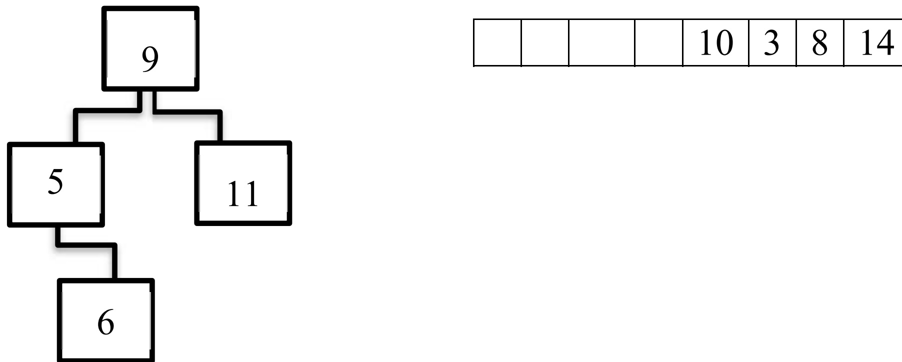


Рисунок 14 – Бинарное дерево на шаге 4

Остальные значения из таблицы добавляются по такому же принципу.

Шаг 5. Результат представлен на рисунке 15.

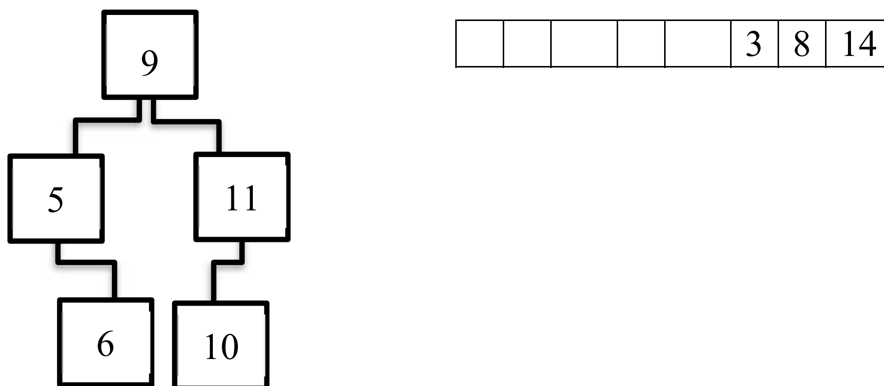


Рисунок 15 – Бинарное дерево на шаге 5

Шаг 6. Результат построения дерева на шаге 6 представлен на рисунке 16.

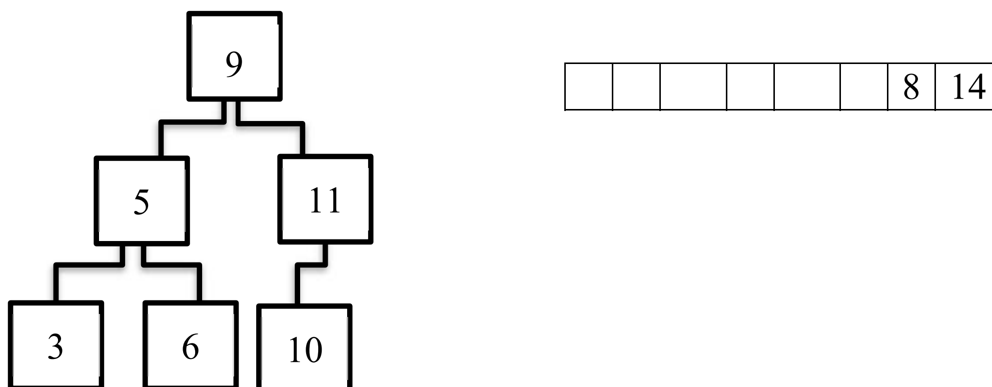


Рисунок 16 – Бинарное дерево на шаге 6

Шаг 7. Достраивается узел 8 (рисунок 17).

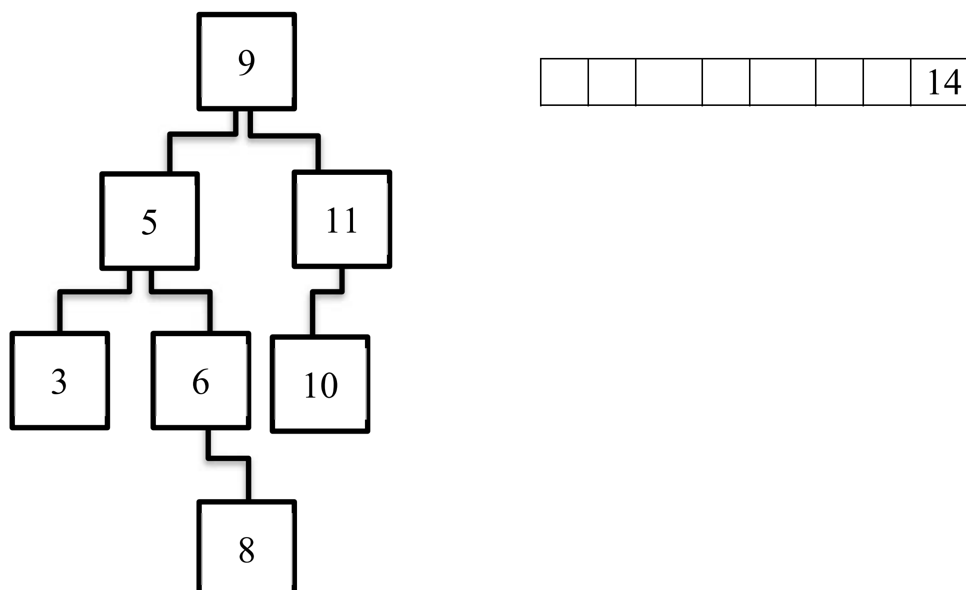


Рисунок 17 – Бинарное дерево на шаге 7

Шаг 8. Построенное бинарное дерево представлено на рисунке 18.

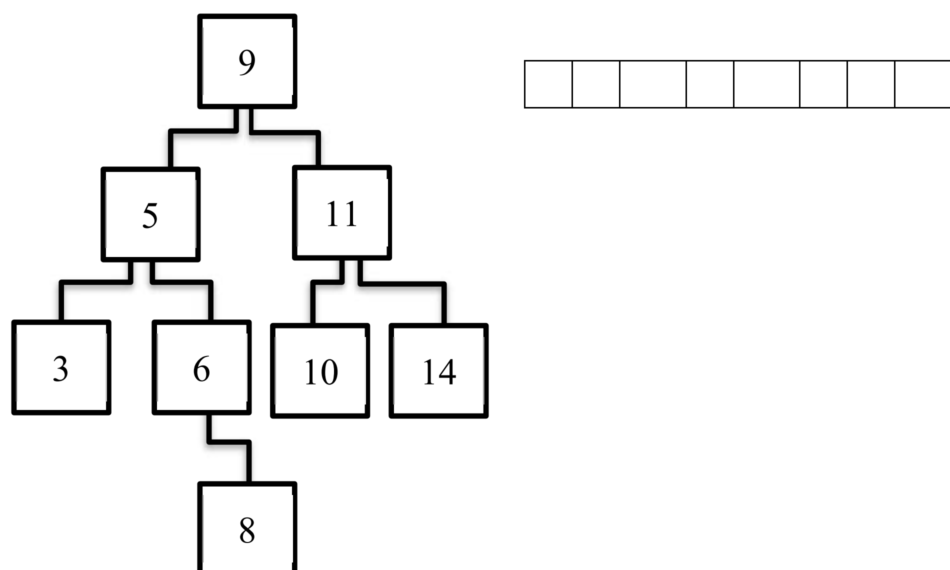
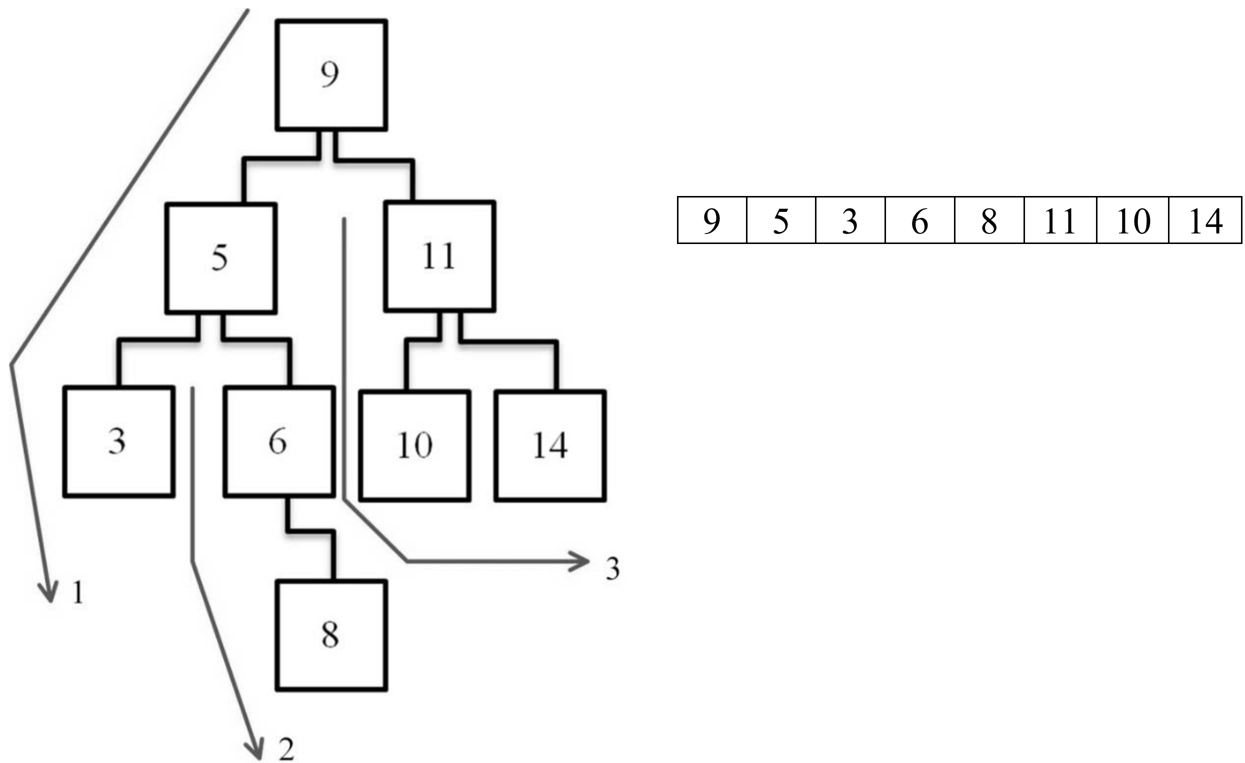


Рисунок 18 – Построенное бинарное дерево

Алгоритмы прохождения всех узлов дерева (обхода).

Прямой обход. При прямом обходе двигаемся сверху вниз. Начинаем с корня дерева 9 и двигаемся влево, проходя все узлы (9 – 5 – 3). Если двигаться больше некуда, возвращаемся на прошлый узел и смотрим можем ли мы с него двигаться вправо, если можем, то переходим на него и продолжаем двигаться по левой стороне (6 – 8), если нет, то возвращаемся ещё на узел назад до тех пор, пока мы не сможем двигаться куда-либо или пока не дойдём до вершины. Таким

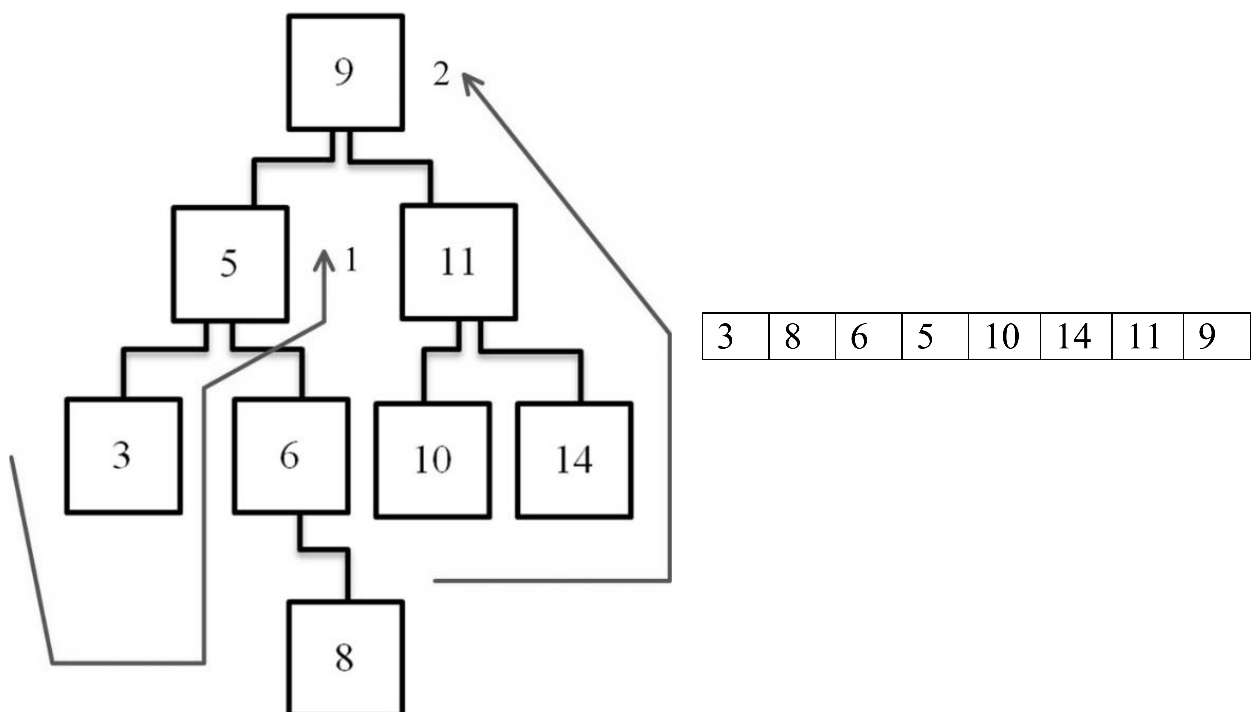
образом, мы обходим левое поддерево. Правое поддерево обходим по такому же принципу (11 – 10 – 14) (рисунок 19).



9	5	3	6	8	11	10	14

Рисунок 19 – Прямой обход дерева

Обратный обход. Суть обратного обхода в том, что прежде, чем посетить дерево, мы должны посетить его поддерево (рисунок 20).

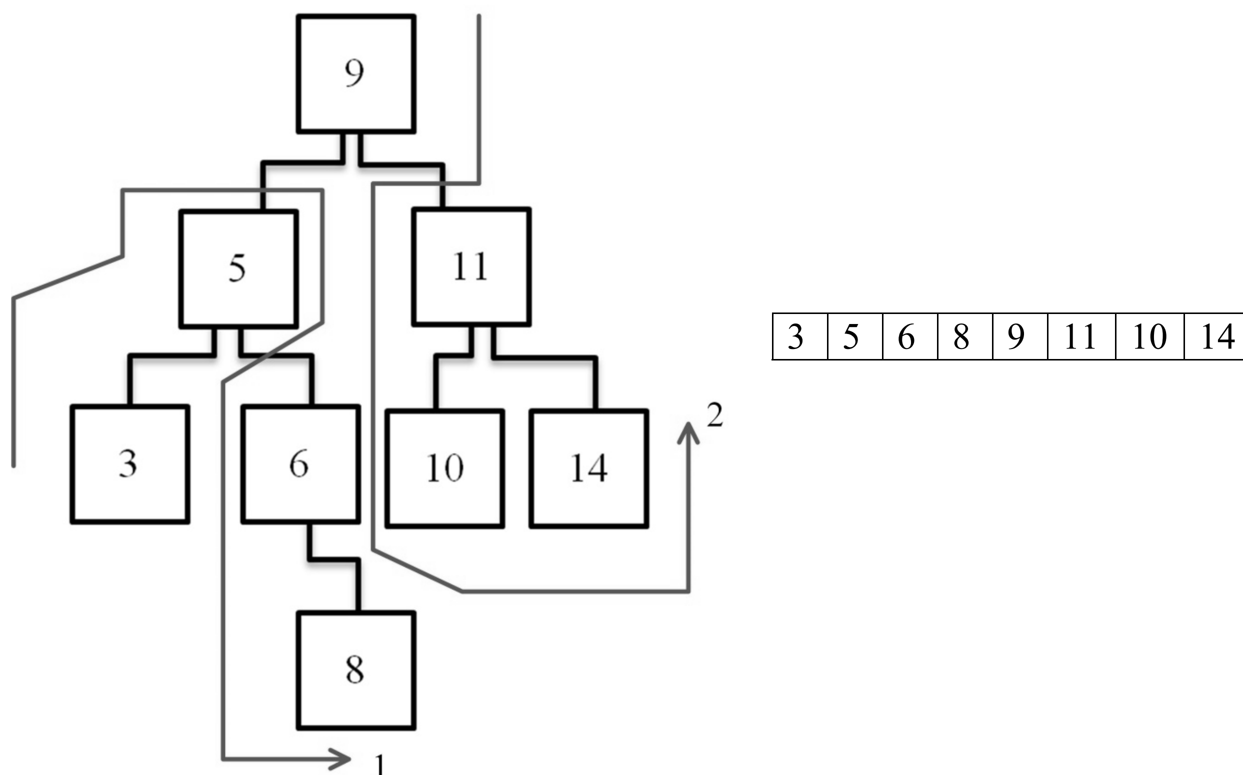


3	8	6	5	10	14	11	9

Рисунок 20 – Обратный обход дерева

В нашем случае начинаем обход с узла 3. Узел 3 – поддерево 5, однако 3 не единственное поддерево, поэтому чтобы пройти 5 мы должны пройти и его правое поддерево, следовательно обходим 8, а затем и 6. Только теперь мы можем прийти в 5. Таким образом мы обошли левое поддерево корня 9, так же обходим и правое поддерево. Только обойдя и правое и левое поддерево мы можем закончить обход, придя к корню (см. рисунок 20).

Симметричный обход. Симметричный обход начинаем с обхода левого поддерева. Начинаем обход с узла 3 и двигаемся к 5, затем обходим правое поддерево пятёрки, т. е. 6, а затем и 8 (рисунок 21).



3	5	6	8	9	11	10	14

Рисунок 21 – Симметричный обход дерева

На этом обход левого поддерева корня завершён, следовательно, мы можем обойти и сам корень дерева. После мы обходим правое поддерево корня, т. е. 11, 10 и 14. Обход завершён (см. рисунок 21).

Реализация бинарного дерева на Python.

```
from binarytree import Node, tree #подключение библиотеки для работы с бинарными деревьями
#задание дерева вручную

root = Node(1) #выбор корневого узла
root.left = Node(2)
root.right = Node(3)
root.left.left = Node(4)
root.left.right = Node(5)
root.right.right = Node(6)
```

```
root.right.left = Node(7)
print(root) #вывод дерева

#обход деревьев с использованием различных алгоритмов
print(root.preorder)
print(root.inorder)
print(root.postorder)
print(root.properties) #вывод свойств 1-го дерева

#задание случайного дерева с 5 уровнями
my_tree = tree(height=5, is_perfect=True)
print(my_tree) #вывод
print(my_tree.properties) #вывод свойств 2-го дерева
```

Практические задания

Дано арифметическое выражение.

Необходимо:

– построить для него бинарное дерево (построение произвести в тетради);

– получить запись арифметического выражения в префиксной и постфиксной форме в соответсвии с вариантом (рисунок 22).

Вариант 1 M+K/(L–N·P)+T·Y	**Вариант 2** ((Y+X·K)·(G–S)+T)·(F–K)+D	**Вариант 3** (T–S/R)·(Z+X+Y/Q)
Вариант 4 (A·B+(C–D)·F)/G–H	**Вариант 5** (M·L–(N+K·F)/S)·T	**Вариант 6** P·(A–Q–S)+L/(D+K)
Вариант 7 F+D–G·(X+Y–Z)	**Вариант 8** K·T·(L–E/R+S)	**Вариант 9** (F·(P–G)+Z/K) D–T
Вариант 10 M–C/L/(S+R·B)	**Вариант 11** (A–P+R/(B–C))/(K+L·F)	**Вариант 12** (((M+N)·R+K)·D+F)/G
Вариант 13 S/P–Y·X+D/F	**Вариант 14** A/(B+C)–D·F+G	**Вариант 15** R+M·(F–K/N)+G·T

Рисунок 22 – Варианты арифметических выражений

Пример 2 – Дано математическое выражение

$$(x+1{,}5)(x-10)^{(x+5)/(x-3{,}1)}.$$

Построить для него бинарное дерево.

Для примера 2 бинарное дерево представлено на рисунке 23.

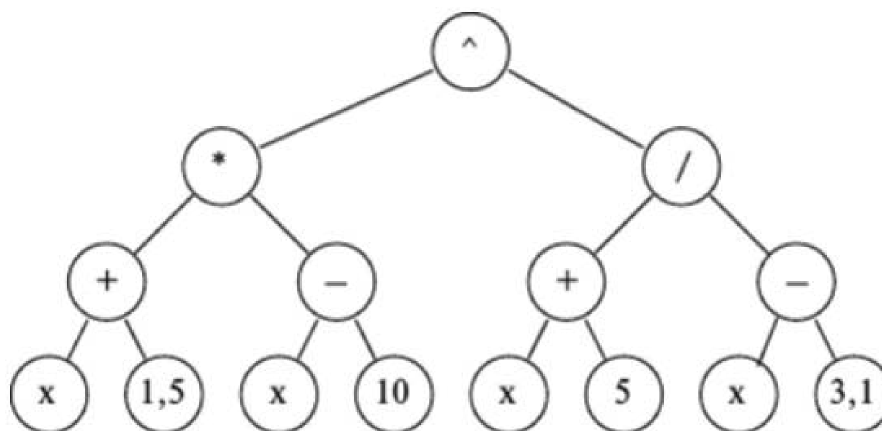


Рисунок 23 – Бинарное дерево для заданного математического выражения

Контрольные вопросы

1 Что представляет собой дерево?
2 Что такое длина и глубина узла?
3 Что такое уровень узла?
4 Какое дерево называется упорядоченным?
5 Какое дерево называется бинарным?
6 Какую структуру имеет бинарное дерево поиска?
7 По каким правилам расположены элементы бинарного дерева поиска?

4 Лабораторная работа № 4. Элементарные алгоритмы для работы с графами

Цель работы: изучить работу и реализацию алгоритма поиска в глубину и алгоритма поиска в ширину.

Основные теоретические положения

Алгоритм поиска в глубину.

Поиск в глубину (***DFS***) – один из методов обхода графа. Стратегия поиска в глубину, как и следует из названия, состоит в том, чтобы идти «вглубь» графа, насколько это возможно. Алгоритм поиска описывается рекурсивно: перебираем все исходящие из рассматриваемой вершины рёбра. Если ребро ведёт в вершину, которая не была рассмотрена ранее, то запускаем алгоритм от этой нерассмотренной вершины, а после возвращаемся и продолжаем перебирать рёбра. Возврат происходит в том случае, если в рассматриваемой вершине не осталось рёбер, которые ведут в нерассмотренную вершину. Если после завершения алгоритма не все вершины были рассмотрены, то необходимо запустить алгоритм от одной из нерассмотренных вершин.

Цель алгоритма состоит в том, чтобы пометить каждую вершину как «пройденная», избегая при этом циклов.

Поскольку мы обходим каждого «соседа» каждого узла, игнорируя тех, которых посещали ранее, мы имеем время выполнения, равное O(|V| + |E|), где |V| – общее количество вершин; |E| – общее количество ребер.

Может показаться, что правильнее использовать |V| · |E|, однако рассмотрим, что означает |V| · |E|.

|V| · |E| означает, что применительно к каждой вершине, мы должны исследовать все ребра графа безотносительно принадлежности этих ребер конкретной вершине. С другой стороны, |V| + |E| означает, что для каждой вершины мы оцениваем лишь примыкающие к ней ребра. Возвращаясь к примеру, каждая вершина имеет определенное количество ребер и, в худшем случае, мы обойдем все вершины (O(|V|)) и исследуем все ребра (O(|E|)). Мы имеем |V| вершин и |E| ребер, поэтому получаем |V| + |E|.

Пример работы алгоритма.

Дан граф (рисунок 24). Найти путь S – t.

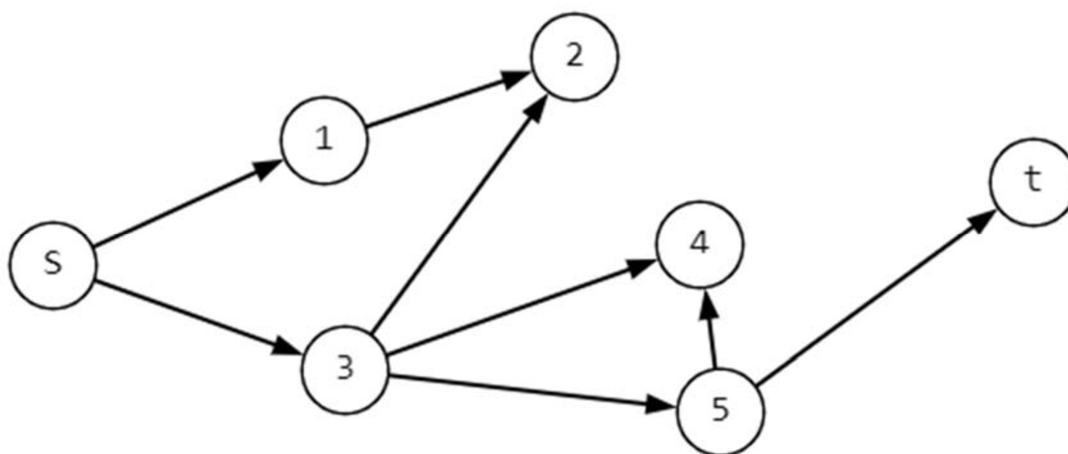


Рисунок 24 – Ориентированный граф

Шаг 1. Из начальной вершины S движемся по пути в смежную вершину, в нашем случае – в вершину 1 (рисунок 25).

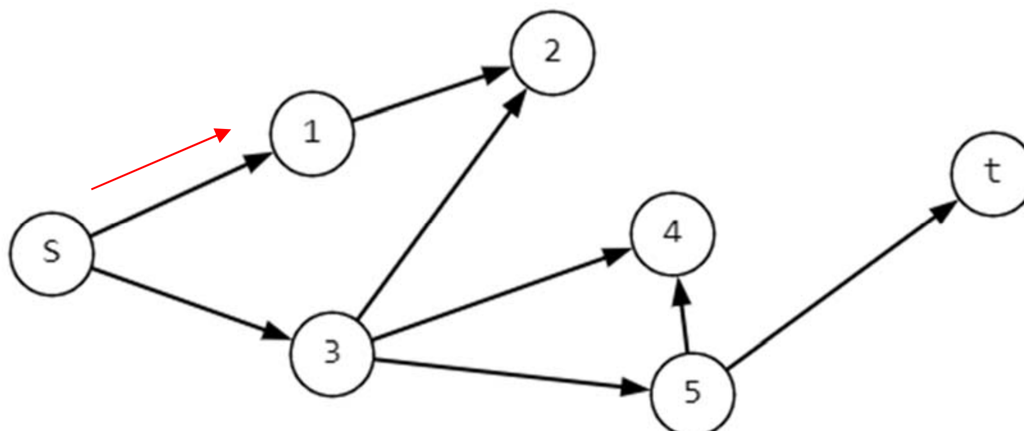


Рисунок 25 – Шаг 1 работы алгоритма

Шаг 2. Из вершины 1 мы движемся в смежную ей вершину 2 (рисунок 26).

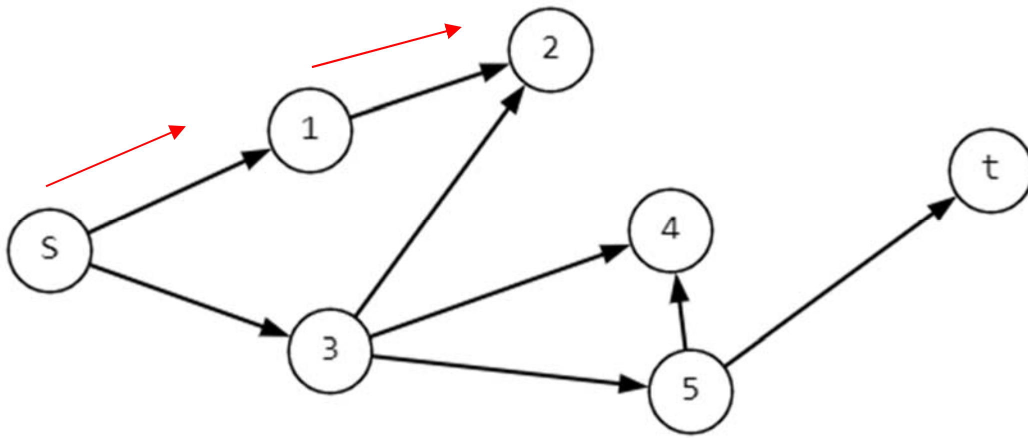


Рисунок 26 – Шаг 2 работы алгоритма

Шаг 3. У вершины 2 нет смежных вершин, поэтому мы возвращаемся в начальную вершину *S* и движемся в вершину 3 (рисунок 27).

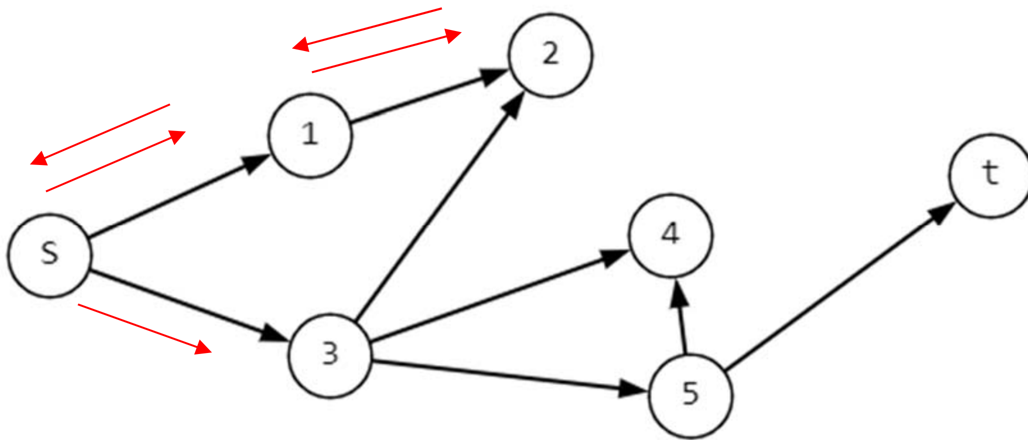


Рисунок 27 – Шаг 3 работы алгоритма

Шаг 4. Из вершины 3 движемся в вершину 4, т. к. в вершине 2 мы уже были (рисунок 28).

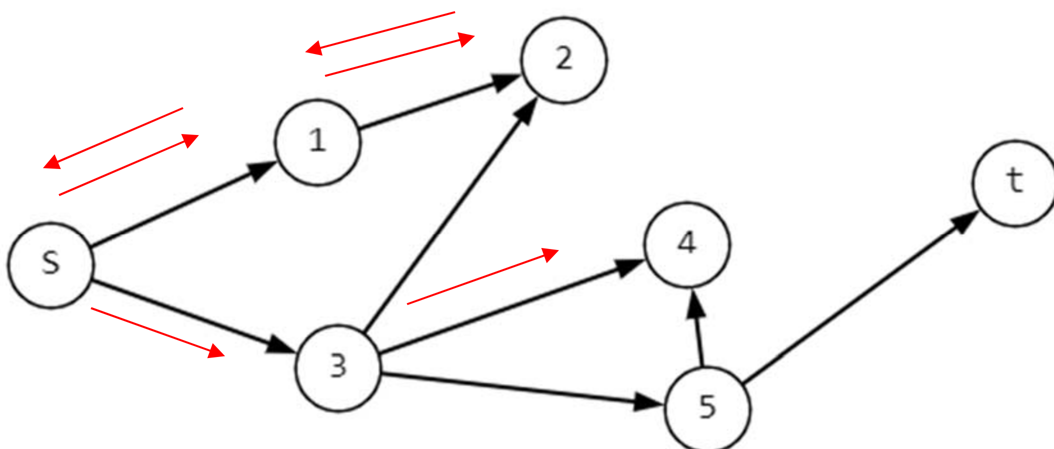


Рисунок 28 – Шаг 4 работы алгоритма

Шаг 5. Из четвёртой вершины мы не можем никуда попасть, поэтому возвращаемся в вершину 3 (рисунок 29).

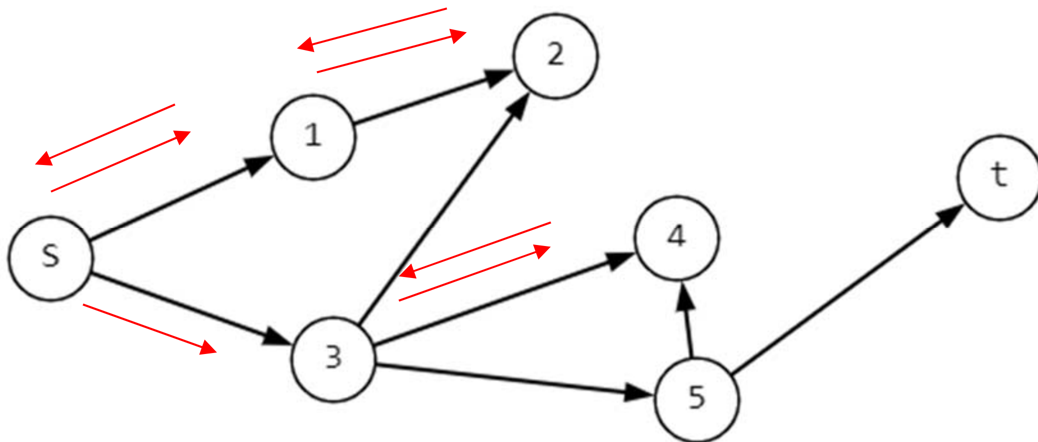


Рисунок 29 – Шаг 5 работы алгоритма

Шаг 6. Из вершины 3 движемся в вершину 5 (рисунок 30).

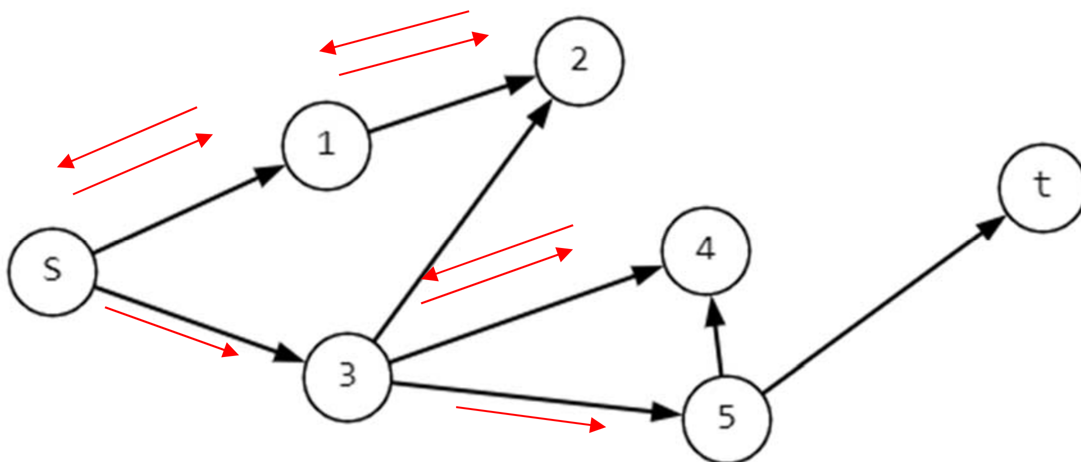


Рисунок 30 – Шаг 6 работы алгоритма

Шаг 7. Поскольку в вершине 4 мы уже были, мы идём в вершину *t* (рисунок 31).

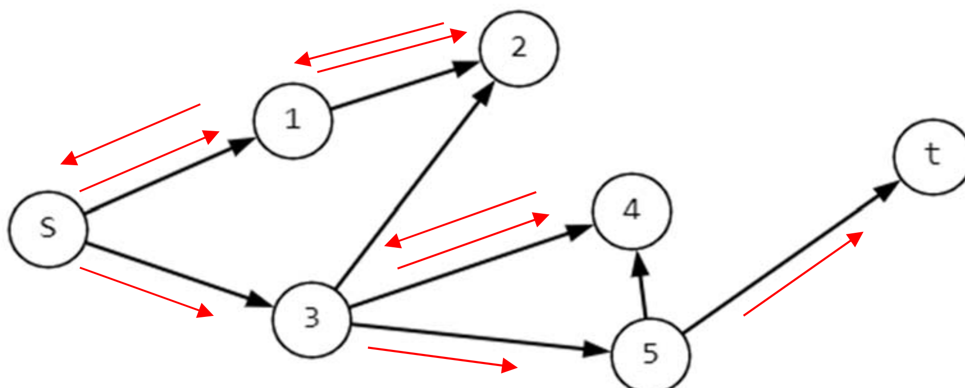
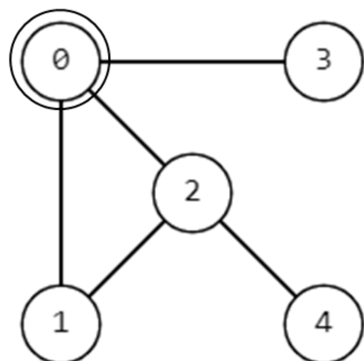


Рисунок 31 – Шаг 7 работы алгоритма

Пример реализации алгоритма.

Шаг 1. Начнем мы с вершины 0. В первую очередь алгоритм поиска в глубину поместит ее саму в список «Пройденные» (на рисунке 31 Visited), а ее смежные вершины – в стек (на рисунке 32 Stack).



Visited

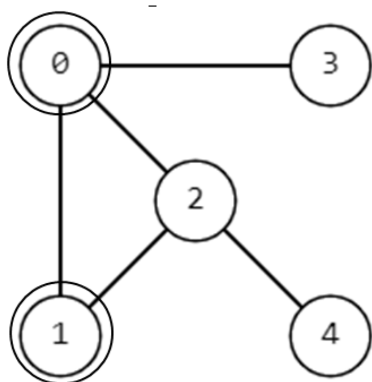
0					

Stack

1	2	3			

Рисунок 32 – Шаг 1 реализации алгоритма поиска в глубину

Шаг 2. Затем мы берем следующий элемент сверху стека, т. е. вершину 1, и переходим к ее соседним вершинам. Поскольку вершина 0 уже пройдена, следующая вершина 2 (рисунок 33).



Visited

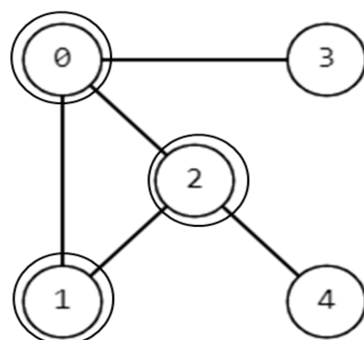
0	1				

Stack

2	3				

Рисунок 33 – Шаг 2 реализации алгоритма поиска в глубину

Шаг 3. Вершина 2 смежна непройденной вершине 4, следовательно мы добавляем ее наверх стека и проходим ее (рисунок 34).



Visited

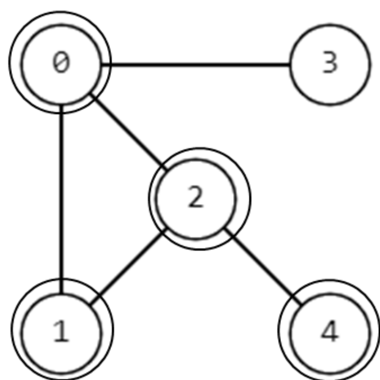
0	1	2			

Stack

4	3				

Рисунок 34 – Шаг 3 реализации алгоритма поиска в глубину

Шаг 4. Проходим вершину 4 (рисунок 35).



Visited

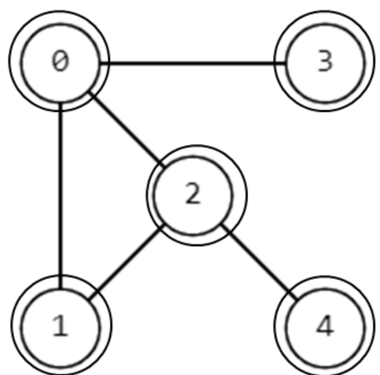
0	1	2	4		

Stack

3					

Рисунок 35 – Шаг 4 реализации алгоритма поиска в глубину

Шаг 5. После того как мы пройдем последний элемент (вершину 3), в стеке не останется непройденных смежных вершин, и таким образом мы завершили обход графа в глубину (рисунок 36).



Visited

0	1	2	4	3	

Stack

Рисунок 36 – Завершение реализации алгоритма поиска в глубину

Код алгоритма.

Псевдокод. Обратите внимание, что в функции init() мы запускаем функцию *DFS* на каждом узле. Это связано с тем, что граф может иметь две разные несвязанные части, поэтому, чтобы убедиться, что мы покрываем каждую вершину, мы также можем запустить алгоритм *DFS* на каждом узле.

```
DFS(G, u)
    u.visited = true
    for each v ∈ G.Adj[u]
        if v.visited == false
            DFS(G,v)

init() {
    For each u ∈ G
        u.visited = false
     For each u ∈ G
       DFS(G, u)
}
```

Реализация на Python.

```
def dfs(graph, start, visited=None):
    if visited is None:
        visited = set()
    visited.add(start)
    print(start)
    for next in graph[start] - visited:
        dfs(graph, next, visited)
    return visited

graph = {'0': set(['1', '2']),
         '1': set(['0', '3', '4']),
         '2': set(['0']),
         '3': set(['1']),
         '4': set(['2', '3'])}

dfs(graph, '0')
```

Алгоритм поиска в ширину.

Обход означает посещение всех узлов графа. ***Обход в ширину* или *Поиск в ширину* (*Breadth first traversal or Breadth first Search*, *BFS*)** – это рекурсивный алгоритм поиска всех вершин графа или древовидной структуры данных.

Алгоритм BFS. Стандартная реализация BFS помещает каждую вершину графа в одну из двух категорий:

1) посещенные;

2) не посещенные.

Цель алгоритма – пометить каждую вершину, как посещенную, избегая циклов.

Алгоритм работает следующим образом.

Шаг 1. Начните с размещения любой вершины графа в конце очереди.

Шаг 2. Возьмите передний элемент очереди и добавьте его в список посещенных.

Шаг 3. Создайте список смежных узлов этой вершины. Добавьте те, которых нет в списке посещенных, в конец очереди.

Шаг 4. Продолжайте повторять шаги 2 и 3, пока очередь не опустеет.

Граф может иметь две разные несвязанные части, поэтому, чтобы убедиться, что мы покрываем каждую вершину, мы также можем запустить алгоритм BFS на каждом узле.

Сложность алгоритма. Так как в памяти хранятся все развёрнутые узлы, пространственная сложность алгоритма составляет $O(|V| + |E|)$.

Полнота. Если у каждого узла имеется конечное число преемников, алгоритм является полным: если решение существует, алгоритм поиска в ширину его находит, независимо от того, является ли граф конечным. Однако если решения не существует, на бесконечном графе поиск не завершается.

Пример BFS. Давайте посмотрим, как алгоритм «поиска в ширину» работает на примере. Мы используем неориентированный граф с пятью вершинами (рисунок 37).

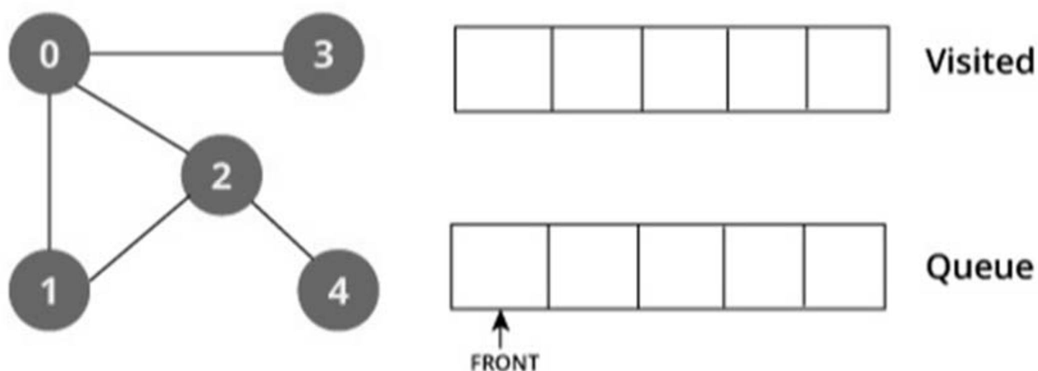


Рисунок 37– Граф для реализации поиска в ширину

Шаг 1. Мы начнем с вершины 0, алгоритм *BFS* начинается с помещения ее в список посещенных и размещения всех смежных вершин в стеке (рисунок 38).

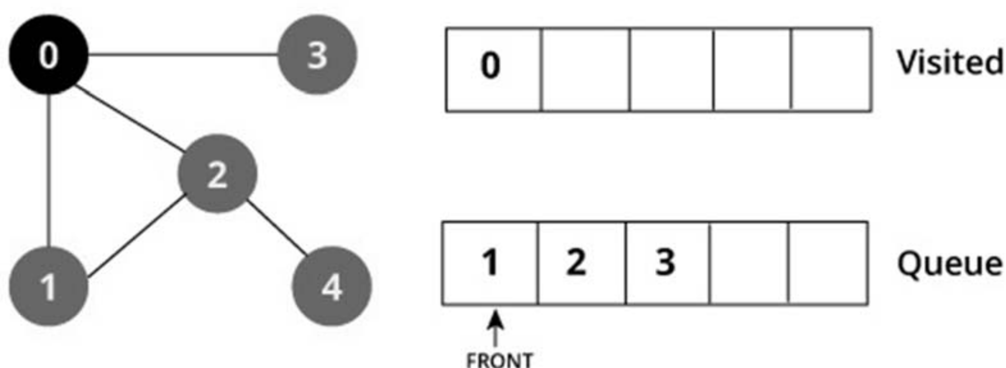


Рисунок 38 – Шаг 1 реализации алгоритма поиска в ширину

Шаг 2. Затем мы посещаем элемент в начале очереди, т. е. 1, и переходим к соседним узлам. Так как 0 уже был посещен, мы посещаем 2 (рисунок 39).

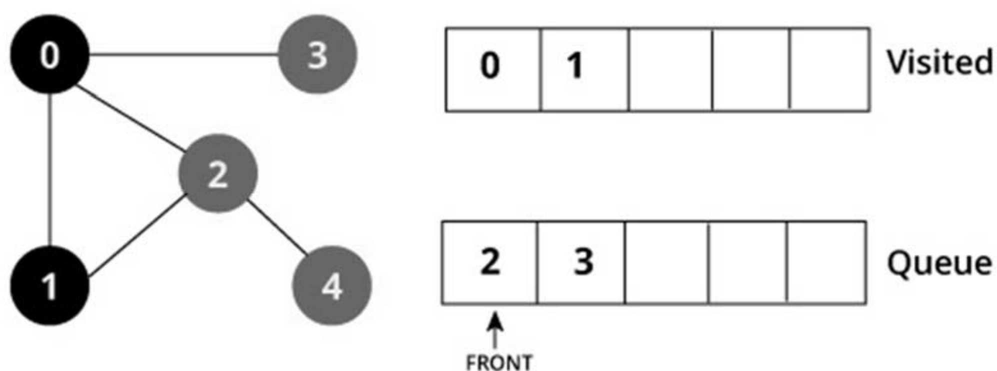


Рисунок 399 – Шаг 2 реализации алгоритма поиска в ширину

Шаг 3. У вершины 2 есть соседняя не посещенная вершина 4, поэтому мы добавляем ее в конец очереди и посещаем 3, которая находится в начале очереди (рисунок 40).

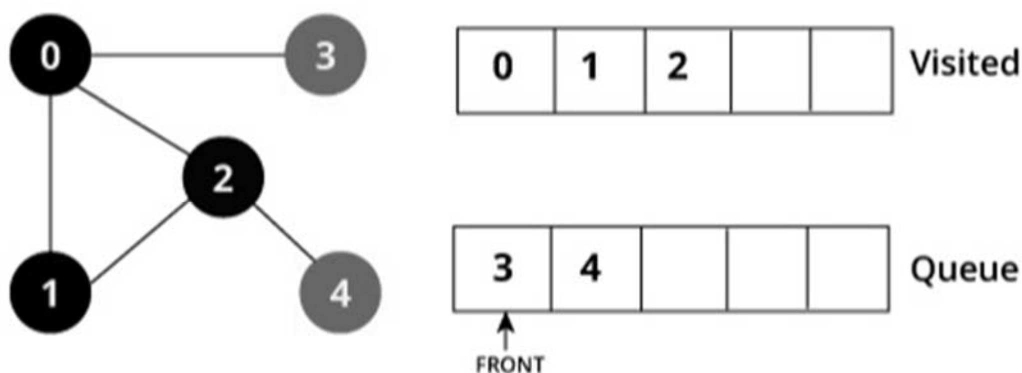


Рисунок 40 – Шаг 3 реализации алгоритма поиска в ширину

Шаг 4. В очереди остается только 4, поскольку единственный соседний узел с 3, т. е. 0, уже посещен. Мы посещаем вершину 4 (рисунок 41).

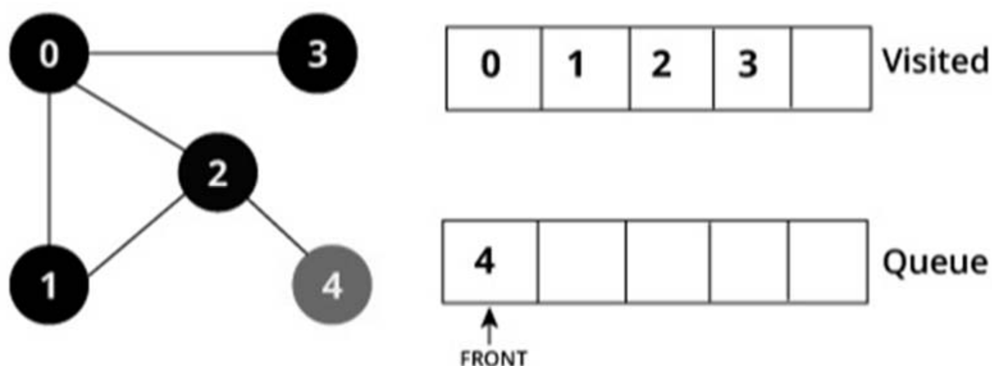


Рисунок 41 – Шаг 4 реализации алгоритма поиска в ширину

Поскольку очередь пуста, мы завершили обход в ширину графа (рисунок 42).

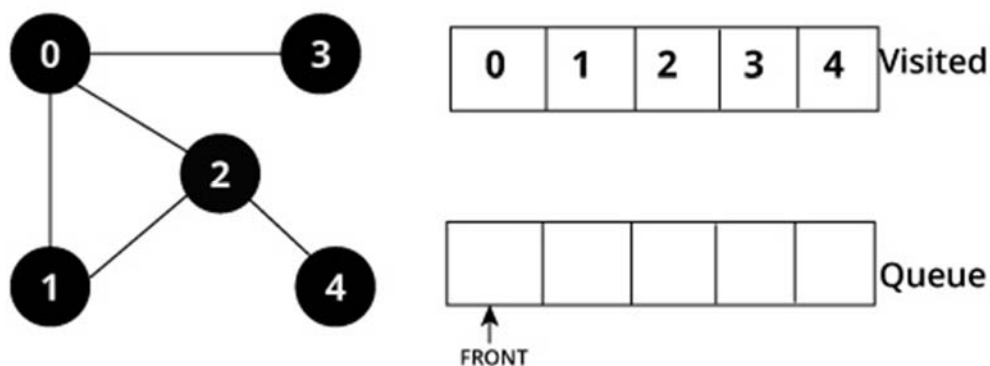


Рисунок 42 – Завершение реализации алгоритма поиска в ширину

Псевдокод.

```
create a queue Q
mark v as visited and put v into Q
while Q is non-empty
   remove the head u of Q
   mark and enqueue all (unvisited) neighbours of u
```

Реализация на Python.

```
import collections
def bfs(graph, root):
   visited, queue = set(), collections.deque([root])
   visited.add(root)
   while queue:
      vertex = queue.popleft()
      for neighbour in graph[vertex]:
         if neighbour not in visited:
            visited.add(neighbour)
            queue.append(neighbour)
if __name__ == '__main__':
   graph = {0: [1, 2], 1: [2], 2: [3], 3: [1,2]}
   breadth_first_search(graph, 0)
and enqueue all (unvisited) neighbours of u
```

Практические задания

1 Реализовать вручную пошагово алгоритм поиска в ширину на заданных графах (рисунок 43).

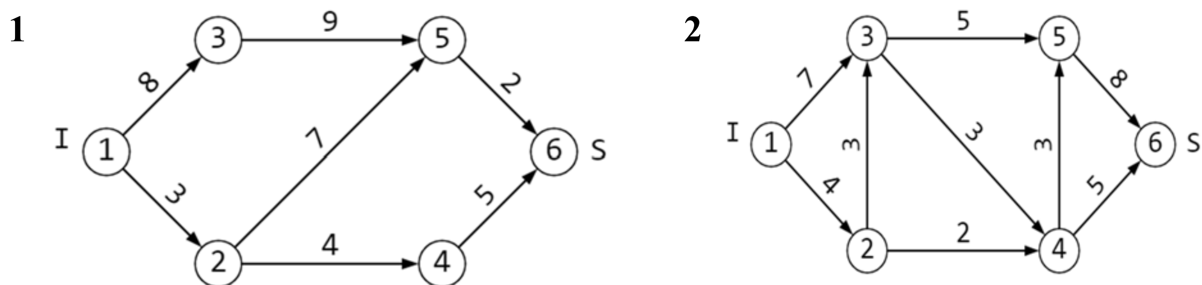


Рисунок 43 – Графы для реализации алгоритма поиска в ширину

2 Реализовать вручную пошагово алгоритм поиска в глубину на заданных графах (рисунок 44).

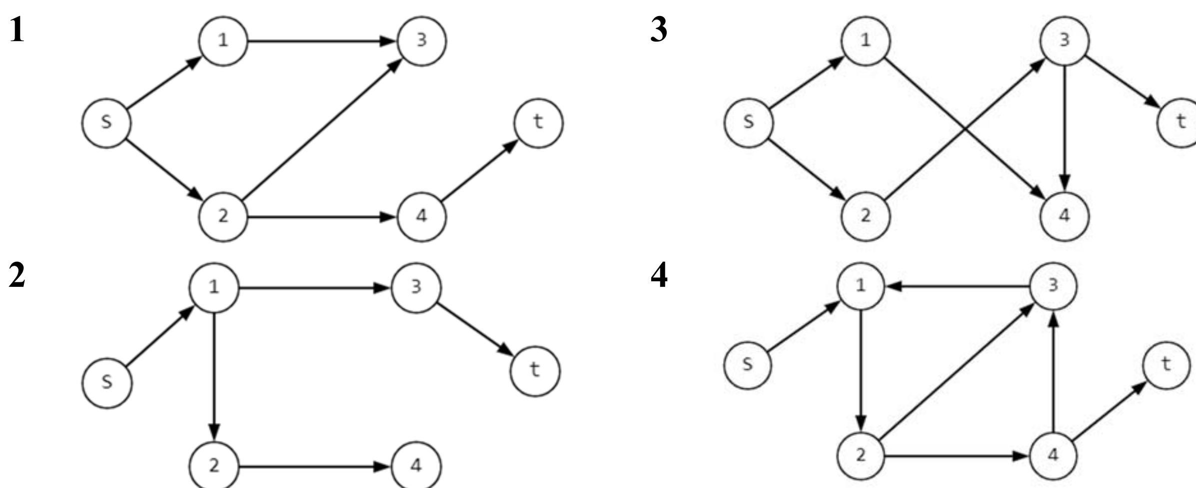


Рисунок 44 – Графы для реализации алгоритма поиска в глубину

Контрольные вопросы

1 Как работает алгоритм поиска в глубину?
2 Назовите условие останова алгоритма поиска в глубину.
3 Как определяется время выполнения поиска в глубину?
4 Как работает алгоритм поиска в ширину?

5 Лабораторная работа № 5. Оптимизационные алгоритмы для работы с графами. Кратчайшие пути в графе

Цель работы: изучить оптимизационные алгоритмы поиска кратчайших путей в графе.

Основные теоретические положения

Стандартные задачи оптимизации в графе заключаются в нахождении пути минимального веса (минимальной длины для не взвешенного графа).

Постановка задачи о кратчайшем пути может быть в следующем виде:

– задача о кратчайшем пути между парой вершин (Single-pair shortest path problem): требуется найти кратчайший путь из заданной вершины *s* в заданную вершину *d*;

– задача о кратчайших путях из заданной вершины во все (Single-source shortest path problem): найти кратчайшие пути из заданной вершины *s* во все;

– задача о кратчайшем пути в заданный пункт назначения (Single-destination shortest path problem): требуется найти кратчайшие пути в заданную вершину *v* из всех вершин графа;

– задача о кратчайшем пути между всеми парами вершин (All-pairs shortest path problem): требуется найти кратчайший путь из каждой вершины *u* в каждую вершину *v*.

Пусть V – множество вершин графа; $|V| = n$; E – множество ребер графа; $|E| = m$.

Алгоритм Дейкстры (1959) используется для поиска кратчайших путей из одной заданной вершины a во все остальные (таблица 2). Вершина a называется источником. Алгоритм способен верно составить решение как для взвешенного, так и для не взвешенного графов, однако не может верно строить пути при наличии отрицательных весов. Время работы $O(n^2 + m)$, однако для разреженных графов алгоритм можно модифицировать, получая время $O(n \log n + m)$ (см. таблицу 2).

Таблица 2 – Отличительные особенности алгоритмов

Алгоритм	Применение
Дейкстры	Находит кратчайший путь от одной из вершин графа до всех остальных. Алгоритм работает только для графов без ребер отрицательного веса
Беллмана – Форда	Находит кратчайшие пути от одной вершины графа до всех остальных во взвешенном графе. Вес ребер может быть отрицательным
Флойда – Уоршелла	Находит кратчайшие пути между всеми вершинами взвешенного ориентированного графа. Сложен в описании, но прост в реализации
Джонсона	Находит кратчайшие пути между всеми парами вершин взвешенного ориентированного графа (должны отсутствовать циклы с отрицательным весом)

Принцип работы алгоритма Дейкстры следующий: каждой вершине из множества V сопоставим метку – минимальное известное расстояние от этой вершины до a. Алгоритм работает пошагово – на каждом шаге он «посещает» одну вершину и пытается уменьшать метки. Работа алгоритма завершается, когда все вершины посещены.

Инициализация. Метка самой вершины a полагается равной 0, метки остальных вершин – бесконечности (в программной реализации за бесконечность принимается число, которое гарантированно превышает вес максимального из путей, или же просто максимальное возможное значение типа данных). Это отражает то, что расстояния от a до других вершин пока неизвестны. Все вершины графа помечаются как не посещённые.

Шаг алгоритма. Если все вершины посещены, алгоритм завершается. В противном случае из ещё не посещённых вершин выбирается вершина u, имеющая минимальную метку длины. Рассматриваем всевозможные маршруты, в которых u является предпоследним пунктом. Вершины, в которые ведут ребра из u, будем звать соседями этой вершины. Для каждого соседа вершины u, кроме помеченных как посещённые, рассмотрим новую длину пути, равную сумме значений текущей метки u и длины ребра, соединяющего u с этим соседом. Если полученное значение длины меньше прежнего значения метки соседа, заменим значение метки полученным значением длины. Рассмотрев всех соседей, пометим вершину u как посещенную и повторим шаг алгоритма.

Псевдокод алгоритма Дейкстры.

```
1   procedure Dijkstra;
2   begin
3     S:= {1};
4     for i:= 2 to n do
5          D[i]:= C[l, i]; { инициализация D }
6     for i:= 1 ton – 1 do begin
7          выбор из множества V\S такой вершины w, что значение D[w] мини-
           мально;
8          добавить w к множеству S;
9          for каждая вершина v из множества V\S do
10            D[v]:= min(D[v], D[w] + C[w, v] )
11    end;
12  end.
```

Здесь предполагается, что в графе G вершины помечены целыми числами, т. е. множество вершин $V = \{1, 2, ..., n\}$, причем вершина 1 является источником. Массив C – это двумерный массив стоимостей, где элемент $C[i, j]$ равен стоимости дуги (i, j). Если дуги (i, j) не существует, то $C[i, j]$ приравнивается к ∞. На каждом шаге массив $D[i]$ содержит длину текущего кратчайшего особого пути к вершине i. S – множество посещенных вершин, его можно организовать как очередь или список. $V \setminus S$ – множество не посещенных вершин графа, его можно реализовать как очередь с приоритетом.

Алгоритм Флойда – динамический алгоритм вычисления значений кратчайших путей для каждой из вершин графа. Метод работает на взвешенных графах, с положительными и отрицательными весами ребер, но без отрицательных циклов, являясь, таким образом, более общим в сравнении с алгоритмом Дейкстры, т. к. последний не работает с отрицательными весами ребер, и к тому же классическая его реализация подразумевает определение оптимальных расстояний от одной вершины до всех остальных.

Для реализации алгоритма Флойда сформируем матрицу смежности D[][] графа G=(V, E), в котором каждая вершина пронумерована от 1 до $|V|$. Эта матрица имеет размер $|V|$x$|V|$, и каждому ее элементу $D[i][j]$ присвоен вес ребра, идущего из вершины i в вершину j. По мере выполнения алгоритма, данная матрица будет перезаписываться: в каждую из ее ячеек внесется значение, определяющее оптимальную длину пути из вершины i в вершину j (отказ от выделения специального массива для этой цели сохранит память и время).

Перед составлением основной части алгоритма необходимо разобраться с содержанием матрицы кратчайших путей. Поскольку каждый ее элемент $D[i][j]$ должен содержать наименьший из имеющихся маршрутов, то сразу можно сказать, что для единичной вершины он равен нулю, даже если она имеет петлю (отрицательные циклы не рассматриваются), следовательно, все элементы главной диагонали $D[i][i]$ нужно обнулить.

А чтобы нулевые недиагональные элементы (матрица смежности могла иметь нули в тех местах, где нет непосредственного ребра между вершинами i и j) сменили по возможности свое значение, определим их равными бесконечности, которая в программе может являться, например, максимально возможной длиной пути в графе, либо просто – большим числом.

Ключевая часть алгоритма, состоя из трех циклов, выражения и условного оператора, записывается довольно компактно:

```
Для k от 1 до |V| выполнять
    Для i от 1 до |V| выполнять
        Для j от 1 до |V| выполнять
Если D[i][k] + D[k][j] < D[i][j] то D[i][j] ← (D[i][k] + D[k][j])
```

Кратчайший путь из вершины i в вершину j может проходить, как только через них самих, так и через множество других вершин $k \in (1, \ldots, |V|)$. Оптимальным из i в j будет путь или не проходящий через k, или проходящий. Заключить о наличии второго случая, значит установить, что такой путь идет из i до k, а затем из k до j, поэтому должно заменить, значение кратчайшего пути $D[i][j]$ суммой $D[i][k] + D[k][j]$.

Пример выполнения алгоритма Флойда.

Дан граф (рисунок 45).

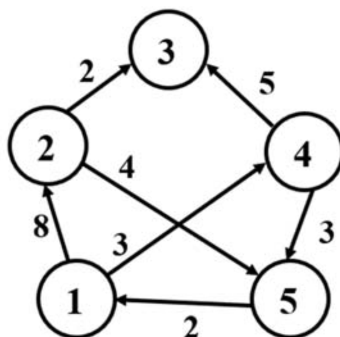


Рисунок 45 – Графы для выполнения алгоритма Флойда

Решение

Составим матрицу графа.

$D^0 =$

	1	2	3	4	5
1	0	8	∞	3	∞
2	∞	0	2	∞	4
3	∞	∞	0	∞	∞
4	∞	∞	5	0	3
5	2	∞	∞	∞	0

Используем формулу $D_{i,j}^{k} = \min(D_{i,j}^{k-1}, D_{i,k}^{k-1} + D_{k,j}^{k-1})$.

$$D_{i,j}^{1} = \min(D_{i,j}^{0}, D_{i,1}^{0} + D_{1,j}^{0}).$$

$D^1 =$

	1	2	3	4	5
1	0	8	∞	3	∞
2	∞	0	2	∞	4
3	∞	∞	0	∞	∞
4	∞	∞	5	0	3
5	2	10	∞	5	0

$$D_{i,j}^{2} = \min(D_{i,j}^{1}, D_{i,2}^{1} + D_{2,j}^{1}).$$

$D^2 =$

	1	2	3	4	5
1	0	8	10	3	12
2	∞	0	2	∞	4
3	∞	∞	0	∞	∞
4	∞	∞	5	0	3
5	2	10	12	5	0

$$D_{i,j}^{3} = \min(D_{i,j}^{2}, D_{i,3}^{2} + D_{3,j}^{2}).$$

$D^3 =$

	1	2	3	4	5
1	0	8	10	3	12
2	∞	0	2	∞	4
3	∞	∞	0	∞	∞
4	∞	∞	5	0	3
5	2	10	12	5	0

$$D_{i,j}^{4} = \min(D_{i,j}^{3}, D_{i,4}^{3} + D_{4,j}^{3}).$$

$D^4 =$

	1	2	3	4	5
1	0	8	8	3	6
2	∞	0	2	∞	4
3	∞	∞	0	∞	∞
4	∞	∞	5	0	3
5	2	10	10	5	0

$$D_{i,j}^5 = \min(D_{i,j}^4, D_{i,5}^4 + D_{5,j}^4).$$

$D^5 =$

	1	2	3	4	5
1	0	8	8	3	6
2	6	0	2	9	4
3	∞	∞	0	∞	∞
4	5	13	5	0	3
5	2	10	10	5	0

Алгоритм Флойда выполнен.

Практические задания

1 Реализовать алгоритм Дейкстры на Python с использованием библиотеки networkx и библиотеки для визуализации графа.

2 Реализовать алгоритм Флойда на Python с использованием библиотеки networkx и библиотеки для визуализации графа.

Контрольные вопросы

1 Какие известны алгоритмы поиска кратчайшего пути в графе?
2 Как формулируется алгоритм Флойда?
3 Когда был разработан алгоритм Дейкстры?
4 Какой принцип работы алгоритма Дейкстры?

6 Лабораторная работа № 6. Оптимизационные алгоритмы для работы с графами. Остовные деревья минимальной стоимости

Цель работы: сформировать знания и умения построения алгоритмов нахождения остовных деревьев минимальной стоимости в графе.

Основные теоретические положения

Пусть задан граф G как совокупность вершин V и ребер E: $G = (V, E)$. Вершины v_1 и v_2 называются смежными, если существует связывающее их ребро. Ребро (u, v) графа G инцидентно своим вершинам u и v. Два ребра, инцидентные одной вершине, называются смежными.

Пусть граф $G = (V, E)$ взвешенный, т. е. задана функция стоимости для ребер $c(u, v)$, связный и неориентированный.

Остовное дерево – это связный подграф без циклов данного связного неориентированного графа, в который входят все его вершины. Минимальное остовное дерево – это остовное дерево данного графа, имеющее минимальный возможный вес. Под весом остовного дерева понимается сумма весов всех ребер, входящих в него.

Граф тогда и только тогда имеет остовное дерево, когда он связен. Если граф не связен, то все его подграфы, у которых множество вершин совпадает с множеством вершин исходного графа, несвязны и не могут содержать дерева.

Если ребра в графе имеют равные веса, минимальное остовное дерево может не быть единственным.

Сечение графа есть разбиение множества всех вершин графа на два непересекающихся множества. Перекрестное ребро – это ребро, которое соединяет вершину одного множества с вершиной другого множества.

Свойство сечения: при любом сечении графа каждое минимальное перекрестное ребро принадлежит некоторому минимальному остовному дереву и каждое минимальное остовное дерево содержит минимальное перекрестное ребро.

Алгоритм Прима находит минимальное остовное дерево (таблица 3). Идея данного алгоритма заключается в добавлении одной за другой вершин в дерево таким образом, чтобы расстояние от уже помеченных вершин до нее было минимальным.

Таблица 3 – Принцип работы алгоритмов

Алгоритм	Способ построения минимального остовного дерева
Прима	Строит минимальное остовное дерево, добавляя на каждом шаге по одному ребру, которое присоединяется к единственному растущему дереву
Краскала	Разобщенный лес поддеревьев постепенно объединяется в единственное дерево

Описание алгоритма Прима нахождения минимального остовного дерева: на входе имеется взвешенный связный неориентированный граф, в нем

выбирается произвольная вершина и находится ребро, инцидентное данной вершине и обладающее наименьшим весом. Найденное ребро и две соединяемые им вершины образуют дерево.

Затем рассматриваются ребра графа, один конец которых – это вершина, входящая в дерево, а другой – нет (перекрестные ребра). Из набора данных ребер выбирается ребро с наименьшим весом.

Найденное на каждом шаге ребро присоединяется к дереву (по свойству сечения). Следовательно, на каждом шаге алгоритма высота формируемого дерева увеличивается на единицу.

Работа алгоритма продолжается до тех пор, пока в дерево не будут включены все вершины данного графа.

На выходе имеем минимальное остовное дерево.

Псевдокод алгоритма Прима.

```
1  procedure Prim( G: граф; var T: множество ребер );
2  begin
3  T:= ∅;
4  U:= {1};
5        while U≠V do begin
6        нахождение ребра (u, v) наименьшей стоимости такого, что u∈U
         и v ∈ V\U;
7        T:= T∪ {(u, v)};
8        U:= U ∪ {v}
9        end;
10    end.
```

Условные обозначения переменных и структур в псевдокоде: *G* – исходный граф; *V* – множество вершин графа *G*; *T* – множество ребер минимального остовного дерева; *U* – множество вершин минимального остовного дерева.

Построение алгоритма Прима начинается с дерева, включающего в себя одну (произвольную) вершину. В течение работы алгоритма дерево разрастается, пока не охватит все вершины исходного графа. На каждом шаге алгоритма к текущему дереву присоединяется самое лёгкое из ребер, соединяющих вершину из построенного дерева и вершину не из дерева.

Алгоритм Крускала достаточно прост в своей идее и реализации. Он заключается в сортировке всех ребер в порядке возрастания длины, и поочередному добавлению их в минимальный остов, если они соединяют различные компоненты связности.

Более формально: пусть мы уже нашли некоторые ребра, входящие в минимальный остов. Утверждается, что среди всех ребер, соединяющих различные компоненты связности, в минимальный остов будет входить ребро с минимальной длиной.

Для реализации алгоритма Крускала необходимо уметь сортировать ребра по возрастанию длины (для этого воспользуемся собственным типом данных) и проверять, соединяет ли ребро две различные компоненты связности.

Практические задания

1 Реализовать алгоритм Прима на Python с использованием библиотеки networkx и библиотеки для визуализации графа.

2 Реализовать алгоритм Крускала с использованием библиотеки networkx и библиотеки для визуализации графа.

Контрольные вопросы и задания

1 Что понимают под минимальным остовным деревом?

2 Приведите пример задачи нахождения минимального остовного дерева.

3 Чем отличаются алгоритмы Крускала и Прима?

4 Что подают на вход алгоритма Прима?

7 Лабораторная работа № 7. Поиск компонент сильной связности графа

Цель работы: изучить компоненты сильной связности и алгоритм их нахождения в графе.

Основные теоретические положения

Дан ориентированный граф G, множество вершин которого V и множество дуг – E. Петли и кратные дуги допускаются. Обозначим через $|V|$ количество вершин графа, через $|E|$ количество дуг.

Неориентированный граф называется **связным**, если все его вершины достижимы из некоторой вершины (эквивалентно, из любой его вершины).

Ориентированный граф называется:

– слабо связным, если соответствующий неориентированный граф является связным;

– сильно связным, если всякая вершина v достижима из любой другой вершины множества V.

Орграф называется **односторонним** (или **односторонне связным**), если для любой пары его вершин по меньшей мере одна достижима из другой.

Компонентой связности неориентированного графа называется максимальный по включению связный подграф.

Компонентой сильной связности называется такое (максимальное по включению) подмножество вершин C, что любые две вершины этого подмножества достижимы друг из друга, т. е. для $\forall u, v \in C$:

$$u \mapsto v, v \mapsto u,$$

где символом $\mapsto$ здесь и далее мы будем обозначать достижимость, т. е. существование пути из первой вершины во вторую.

Мостом в графе называется ребро, удаление которого увеличивает число

компонент связности.

Шарниром в графе называется вершина, удаление которой увеличивает число компонент связности.

Связный граф называется **вершинно-*k*-связным** (или просто *k*-связным), если он имеет более *k* вершин и после удаления любых *k*−1 из них остаётся связным. Максимальное такое число *k* называется **вершинной связностью** (или просто **связностью**) графа. 1-связный граф называется связным, 2-связный граф – двусвязным.

Связный граф называется **рёберно-*k*-связным**, если он остаётся связным после удаления любых *k*−1 рёбер. Максимальное такое число *k* называется **рёберной связностью** графа.

Компонентой двусвязности графа называется максимальный по включению двусвязный подграф.

Понятно, что компоненты сильной связности для данного графа не пересекаются, т. е. фактически это разбиение всех вершин графа. Отсюда логично определение **конденсации** G^{SCC} как графа, получаемого из данного графа сжатием каждой компоненты сильной связности в одну вершину. Каждой вершине графа конденсации соответствует компонента сильной связности графа *G*, а ориентированная дуга между двумя вершинами C_i и C_j графа конденсации проводится, если найдётся пара вершин, между которыми существовала дуга в исходном графе, т. е.

$$(u, v) \in E \mid u \in C_i, v \in C_j .$$

Важнейшим свойством графа конденсации является то, что он **ацикличен**.

Описываемый ниже алгоритм выделяет в данном графе все компоненты сильной связности. Построить по ним граф конденсации не составит труда.

Алгоритм Косарайю.

Метод Косарайю прост для реализации и понимания. Так как компоненты сильной связности есть циклы, то они совпадают и у исходного графа, и у его инвертирования.

Пусть дан ориентированный граф *G* = (*V*, *E*). Через *G*' = (*V*, *E*') обозначим инвертирование графа *G*.

Инвертирование – процедура, в ходе которой поменяем направление каждого ребра на противоположное.

Обход графа – это переход от одной его вершины к другой в поисках свойств связей этих вершин. Связи (линии, соединяющие вершины) называются направлениями, путями, гранями или ребрами графа. Вершины графа также именуются узлами.

Такт *DFS* из вершины *V* – обход (*в глубину*) всех вершин графа, достижимых из *V*. Такт можно интерпретировать как рекурсивный вызов функции. Такт обработки вершины, у которой нет соседей, будет равняться 1.

Время выхода вершины – число, соответствующее времени выхода рекурсии алгоритма *DFS* из вершины. Изначально счётчик времени 0, увеличивается

лишь в двух случаях:

1) начало нового такта *DFS*;

2) прохождение по ребру (при том не важно, рекурсивный проход или нет).

Практическое задание

Реализовать алгоритм Косарайю на Python с использованием библиотеки networkx и библиотеки для визуализации графа.

Контрольные вопросы

1 Что такое инвертирование графа?
2 Какой орграф называют сильно связным?
3 Что называется мостом в графе?
4 Что называют компонентами сильной связности орграфа?

8 Лабораторная работа № 8. Задача о максимальном потоке

Цель работы: изучить алгоритм Форда – Фалкерсона для решения задачи о максимальном потоке.

Основные теоретические положения

Задача о максимальном потоке заключается в нахождении такого потока по транспортной сети, что сумма потоков из истока, или, что то же самое, сумма потоков в сток максимальна.

Пример практической задачи о максимальном потоке. Пусть имеется сеть трубопроводов, соединяющих пункт A с пунктом B. Трубопроводы могут соединяться и разветвляться в промежуточных пунктах. Количество ресурса, которое может быть перекачено по каждому отрезку трубопровода в единицу времени не безгранично, и определяется диаметром трубы, мощностью насоса и другими параметрами. Вопрос: сколько ресурса можно прокачать через эту сеть трубопровода в единицу времени?

Пусть $G(V,E)$ – сеть с двумя особыми вершинами: исток и сток. Дуги сети нагружены неотрицательными вещественными числами и для сети строится *матрица пропускных способностей*.

Сеть представляет собой ориентированный граф, в котором каждая дуга имеет положительную пропускную способность.

Исток – вершина с нулевой полустепенью захода.

Сток – вершина с нулевой полустепенью исхода.

Потоком в сети называется некоторая функция, которая ставит в соответствие дуге некоторое число – вес дуги.

Сквозные пути – по которым проходит поток на текущей итерации.

Пропускная способность дуги – максимальное количество ресурса, которое может быть передано по дуге в единицу времени. По дугам сети можно пропустить некоторый поток по пути из истока в сток.

Множество вещественных чисел будем формулировать следующим образом:

$$C : E \to R^+ .$$

Числа c будем называть пропускными способностями дуги:

$$c_{ij} = \begin{cases} c_{ij} \mid \exists\,(i, j); \\ 0 \mid \neg\exists\,(i, j). \end{cases}$$

Мощность потока по пути – количество ресурса, пропускаемого по заданному пути в единицу времени. Мощность потока будет определяться выражением

$$|\mu_k| = \min_{\forall (i,j) \in \mu_k} c_{ij} .$$

Остаточная пропускная способность ребра – значение, которое определяет, сколько ещё потока можно направить из вершины *u* в вершину *v*.

Дуга, по которой пропущено количество ресурса в единицу времени, совпадающего с пропускной способностью этой дуги, называется **насыщенной.**

Разрез – множество рёбер, образующих двудольный подграф, удаление которых делит граф на два или более компонента, которые, в частности, могут быть изолированными узлами.

Алгоритм поиска максимального потока.

Следует понимать *остаточную сеть* как тот же граф, который имеется на входе, но в этом случае мы будем производить над ним некоторые операции.

1 Отправлять определенное количество потока из текущей вершины в соседние.

2 Возвращать определенное количество потока из соседних вершин в текущую.

3 В начальный момент времени поток, который мы хотим провести через нашу сеть, должен быть равен нулю. *Остаточная сеть* совпадает с исходной сетью.

4 Находим любой путь из истока в сток в остаточной сети. Если путь не находим, утверждается, что поток является максимальным.

5 Пускаем через найденный путь поток, равный минимальному весу ребра, которое входит в множество рёбер найденного пути.

6 Из веса рёбер на этом пути высчитываем размер потока, который мы пустили.

7 К весу обратных рёбер (будем считать, что они существуют в *остаточной сети* и равны 0) прибавляем размер потока. Другими словами, на предыдущем шаге мы отправили некоторое количество потока из текущей вершины в следующую, а теперь при желании можем вернуть это же количество потока обратно в текущую.

8 Возвращаемся обратно к нахождению пути в *остаточной сети* после модификации.

Практические задания

1 Реализовать алгоритм Форда – Фалкерсона на Python с использованием библиотеки networkx и библиотеки для визуализации графа.

2 Сформировать на сети заданных графов поток максимальной мощности (рисунок 46).

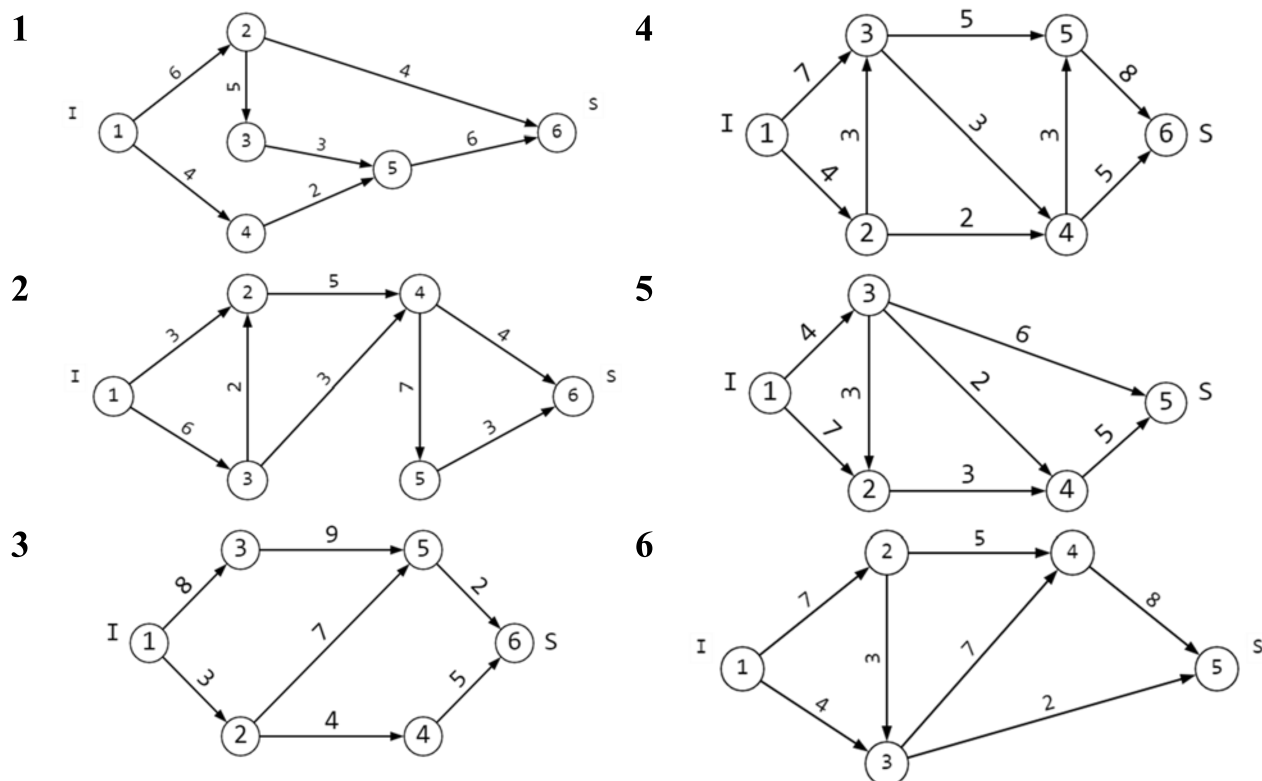


Рисунок 46 – Графы для задачи о максимальном потоке

Контрольные вопросы

1 В чём заключается задача максимального потока?
2 Что такое разрез и пропускная способность разреза?
3 В чем суть алгоритма Форда – Фалкерсона?
4 Как строится матрица пропускной способности?

Список литературы

1 **Микони, С. В.** Дискретная математика для бакалавра: множества, отношения, функции, графы : учебное пособие / С. В. Микони. – Санкт-Петербург; Москва; Краснодар : Лань, 2021. – 192 с. : ил.

2 **Таранников, Ю. В.** Дискретная математика. Задачник : учебное пособие для вузов / Ю. В. Таранников. – Москва: Юрайт, 2020. – 385 с.

3 **Бабичева, И. В.** Дискретная математика. Контролирующие материалы к тестированию : учебное пособие / И. В. Бабичева. – 2-е изд., испр. – Санкт-Петербург; Москва; Краснодар: Лань, 2021. – 160 с. : ил.