**Лабораторная работа 6. АЛГОРИТМЫ НА ГРАФАХ**

**ЦЕЛЬ РАБОТЫ:** Освоить сущность и программную реализацию: а) способов представления графов; б) алгоритмов поиска в ширину и глубину; в) алгоритма топологической сортировки графов. Разобрать алгоритм Прима и алгоритм Крускала

**Теоретические сведения**

**Граф –** математическая модель, с помощью которой удобно представлять бинарное отношение.

Граф можно представить несколькими способами:

* Матрицей смежности;
* Матрицей инцидентности;
* Списком смежных вершин;

Работа с графами подразумевает под собой в основном поиск или сортировку. Рассмотрим некоторые алгоритмы.

Идея **поиска в ширину** состоит в том, чтобы посещать вершины в порядке их удаленности от некоторой заранее выбранной или указанной стартовой вершины. Иначе говоря, сначала посещается сама вершина, затем все вершины, смежные с ней, то есть находящиеся от нее на расстоянии 1, затем вершины, находящиеся от на расстоянии 2, и т.д.

Поиск в ширину был формально предложен [Э. Ф. Муром](https://ru.wikipedia.org/wiki/%D0%9C%D1%83%D1%80,_%D0%AD%D0%B4%D0%B2%D0%B0%D1%80%D0%B4_%D0%A4%D0%BE%D1%80%D0%B5%D1%81%D1%82) в контексте поиска пути в лабиринте. Ли независимо открыл тот же алгоритм в контексте [разводки проводников на печатных платах](https://ru.wikipedia.org/wiki/%D0%90%D0%BB%D0%B3%D0%BE%D1%80%D0%B8%D1%82%D0%BC_%D0%9B%D0%B8).

Поиск в ширину может применяться для решения задач, связанных с [теорией графов](https://ru.wikipedia.org/wiki/%D0%A2%D0%B5%D0%BE%D1%80%D0%B8%D1%8F_%D0%B3%D1%80%D0%B0%D1%84%D0%BE%D0%B2):

* [Волновой алгоритм](https://ru.wikipedia.org/wiki/%D0%92%D0%BE%D0%BB%D0%BD%D0%BE%D0%B2%D0%BE%D0%B9_%D0%B0%D0%BB%D0%B3%D0%BE%D1%80%D0%B8%D1%82%D0%BC) [поиска пути](https://ru.wikipedia.org/wiki/%D0%9F%D0%BE%D0%B8%D1%81%D0%BA_%D0%BF%D1%83%D1%82%D0%B8) в лабиринте
* Волновая [трассировка печатных плат](https://ru.wikipedia.org/wiki/%D0%A2%D1%80%D0%B0%D1%81%D1%81%D0%B8%D1%80%D0%BE%D0%B2%D0%BA%D0%B0_%D0%BF%D0%B5%D1%87%D0%B0%D1%82%D0%BD%D1%8B%D1%85_%D0%BF%D0%BB%D0%B0%D1%82)
* Поиск [компонент связности](https://ru.wikipedia.org/wiki/%D0%9A%D0%BE%D0%BC%D0%BF%D0%BE%D0%BD%D0%B5%D0%BD%D1%82%D0%B0_%D1%81%D0%B2%D1%8F%D0%B7%D0%BD%D0%BE%D1%81%D1%82%D0%B8_%D0%B3%D1%80%D0%B0%D1%84%D0%B0) в графе
* [Поиск кратчайшего пути](https://ru.wikipedia.org/wiki/%D0%97%D0%B0%D0%B4%D0%B0%D1%87%D0%B0_%D0%BE_%D0%BA%D1%80%D0%B0%D1%82%D1%87%D0%B0%D0%B9%D1%88%D0%B5%D0%BC_%D0%BF%D1%83%D1%82%D0%B8) между двумя узлами невзвешенного графа
* [Поиск в пространстве состояний](https://ru.wikipedia.org/wiki/%D0%9F%D0%BE%D0%B8%D1%81%D0%BA_%D0%B2_%D0%BF%D1%80%D0%BE%D1%81%D1%82%D1%80%D0%B0%D0%BD%D1%81%D1%82%D0%B2%D0%B5_%D1%81%D0%BE%D1%81%D1%82%D0%BE%D1%8F%D0%BD%D0%B8%D0%B9): нахождение решения задачи с наименьшим числом ходов, если каждое состояние системы можно представить вершиной графа, а переходы из одного состояния в другое — рёбрами графа
* Нахождение кратчайшего цикла в ориентированном невзвешенном графе
* Нахождение всех вершин и рёбер, лежащих на каком-либо кратчайшем пути между двумя вершинами � �
* Поиск увеличивающего пути в [алгоритме Форда-Фалкерсона](https://ru.wikipedia.org/wiki/%D0%A4%D0%BE%D1%80%D0%B4%D0%B0-%D0%A4%D0%B0%D0%BB%D0%BA%D0%B5%D1%80%D1%81%D0%BE%D0%BD%D0%B0_%D0%B0%D0%BB%D0%B3%D0%BE%D1%80%D0%B8%D1%82%D0%BC) ([алгоритм Эдмондса-Карпа](https://ru.wikipedia.org/wiki/%D0%90%D0%BB%D0%B3%D0%BE%D1%80%D0%B8%D1%82%D0%BC_%D0%AD%D0%B4%D0%BC%D0%BE%D0%BD%D0%B4%D1%81%D0%B0-%D0%9A%D0%B0%D1%80%D0%BF%D0%B0))

Идея **поиска в глубину** заключается в том, что мы двигаемся от начальной вершины (точки, места) в определенном направлении (по определенному пути) до тех пор, пока не достигнем конца пути или пункта назначения (искомой вершины). Если мы достигли конца пути, но он не является пунктом назначения, то мы возвращаемся назад (к точке разветвления или расхождения путей) и идем по другому маршруту.

Поиск в глубину ограниченно применяется как собственно *поиск*, чаще всего на древовидных структурах: когда расстояние между точками малó, поиск в глубину может «плутать» где-то далеко.

Зато поиск в глубину — хороший инструмент для исследования топологических свойств графов. Например:

* В качестве подпрограммы в алгоритмах поиска одно- и [двусвязных](https://ru.wikipedia.org/wiki/%D0%90%D0%BB%D0%B3%D0%BE%D1%80%D0%B8%D1%82%D0%BC_%D0%9A%D0%BE%D1%81%D0%B0%D1%80%D0%B0%D0%B9%D1%8E) компонент.
* В [топологической сортировке](https://ru.wikipedia.org/wiki/%D0%A2%D0%BE%D0%BF%D0%BE%D0%BB%D0%BE%D0%B3%D0%B8%D1%87%D0%B5%D1%81%D0%BA%D0%B0%D1%8F_%D1%81%D0%BE%D1%80%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%BA%D0%B0).
* Для поиска [точек сочленения](https://ru.wikipedia.org/wiki/%D0%A2%D0%BE%D1%87%D0%BA%D0%B0_%D1%81%D0%BE%D1%87%D0%BB%D0%B5%D0%BD%D0%B5%D0%BD%D0%B8%D1%8F_%D0%B3%D1%80%D0%B0%D1%84%D0%B0), [мостов](https://ru.wikipedia.org/wiki/%D0%9C%D0%BE%D1%81%D1%82_(%D1%82%D0%B5%D0%BE%D1%80%D0%B8%D1%8F_%D0%B3%D1%80%D0%B0%D1%84%D0%BE%D0%B2)).
* Для преобразования [синтаксического дерева](https://ru.wikipedia.org/wiki/%D0%90%D0%B1%D1%81%D1%82%D1%80%D0%B0%D0%BA%D1%82%D0%BD%D0%BE%D0%B5_%D1%81%D0%B8%D0%BD%D1%82%D0%B0%D0%BA%D1%81%D0%B8%D1%87%D0%B5%D1%81%D0%BA%D0%BE%D0%B5_%D0%B4%D0%B5%D1%80%D0%B5%D0%B2%D0%BE) в строку (любую: префиксную, [инфиксную](https://ru.wikipedia.org/wiki/%D0%98%D0%BD%D1%84%D0%B8%D0%BA%D1%81%D0%BD%D0%B0%D1%8F_%D0%B7%D0%B0%D0%BF%D0%B8%D1%81%D1%8C), [обратную польскую](https://ru.wikipedia.org/wiki/%D0%9E%D0%B1%D1%80%D0%B0%D1%82%D0%BD%D0%B0%D1%8F_%D0%BF%D0%BE%D0%BB%D1%8C%D1%81%D0%BA%D0%B0%D1%8F_%D0%B7%D0%B0%D0%BF%D0%B8%D1%81%D1%8C)).
* В различных расчётах на графах. Например, как часть [алгоритма Диница](https://ru.wikipedia.org/wiki/%D0%90%D0%BB%D0%B3%D0%BE%D1%80%D0%B8%D1%82%D0%BC_%D0%94%D0%B8%D0%BD%D0%B8%D1%86%D0%B0) поиска максимального потока.

Поиск в глубину — естественный выбор, когда агент (человек или робот) лично ходит по лабиринту и видит то, что непосредственно рядом с ним. «Правило левой руки» (идти, ведя левой рукой по стенке) будет поиском в глубину, если лабиринт древовидный (нет кружных путей).

Задача**топологической сортировки** графа состоит в следующем: указать такой линейный порядок на его вершинах, чтобы любое ребро вело от вершины с меньшим номером к вершине с большим номером. Очевидно, что если в графе есть циклы, то такого порядка не существует.

При помощи топологической сортировки строится корректная последовательность выполнения действий, всякое из которых может зависеть от другого: последовательность прохождения учебных курсов студентами, установки программ при помощи [пакетного менеджера](https://ru.wikipedia.org/wiki/%D0%9F%D0%B0%D0%BA%D0%B5%D1%82%D0%BD%D1%8B%D0%B9_%D0%BC%D0%B5%D0%BD%D0%B5%D0%B4%D0%B6%D0%B5%D1%80), сборки исходных текстов программ при помощи [Makefile](https://ru.wikipedia.org/wiki/Makefile)'ов.

Можно построить список отображения объектов в [изометрической проекции](https://ru.wikipedia.org/wiki/%D0%98%D0%B7%D0%BE%D0%BC%D0%B5%D1%82%D1%80%D0%B8%D1%87%D0%B5%D1%81%D0%BA%D0%B0%D1%8F_%D0%BF%D1%80%D0%BE%D0%B5%D0%BA%D1%86%D0%B8%D1%8F) зная парные порядковые отношения между объектами (какой из двух объектов должен быть прорисован раньше).

**Минимальное остовное дерево** — это подмножество рёбер графа, которое соединяет все вершины, имеющие минимальную сумму весов рёбер, и без циклов.

Существует несколько алгоритмов для нахождения минимального остовного дерева. Некоторые наиболее известные из них перечислены ниже:

* [Алгоритм Прима](https://ru.wikipedia.org/wiki/%D0%90%D0%BB%D0%B3%D0%BE%D1%80%D0%B8%D1%82%D0%BC_%D0%9F%D1%80%D0%B8%D0%BC%D0%B0),
* [Алгоритм Краскала](https://ru.wikipedia.org/wiki/%D0%90%D0%BB%D0%B3%D0%BE%D1%80%D0%B8%D1%82%D0%BC_%D0%9A%D1%80%D0%B0%D1%81%D0%BA%D0%B0%D0%BB%D0%B0) (или алгоритм Крускала),
* [Алгоритм Борувки](https://ru.wikipedia.org/wiki/%D0%90%D0%BB%D0%B3%D0%BE%D1%80%D0%B8%D1%82%D0%BC_%D0%91%D0%BE%D1%80%D1%83%D0%B2%D0%BA%D0%B8),
* [Алгоритм обратного удаления](https://ru.wikipedia.org/wiki/%D0%90%D0%BB%D0%B3%D0%BE%D1%80%D0%B8%D1%82%D0%BC_%D0%BE%D0%B1%D1%80%D0%B0%D1%82%D0%BD%D0%BE%D0%B3%D0%BE_%D1%83%D0%B4%D0%B0%D0%BB%D0%B5%D0%BD%D0%B8%D1%8F) (получение минимального остовного дерева из связного рёберно взвешенного графа).

Алгоритмы Прима и Крускала относятся к **локально-оптимальным** («**жадным**») алгоритмам.

**Алгоритм Крускала**: В начале текущее множество рёбер устанавливается пустым. Затем, пока это возможно, проводится следующая операция: из всех рёбер, добавление которых к уже имеющемуся множеству не вызовет появление в нём цикла, выбирается ребро минимального веса и добавляется к уже имеющемуся множеству. Когда таких рёбер больше нет, алгоритм завершён. Подграф данного графа, содержащий все его вершины и найденное множество рёбер, является его остовным деревом минимального веса.

***Задание 1.*** Ориентированный граф **G** взять в соответствии с вариантом. Представить его в отчете в виде матрицы смежности, матрицы инцидентности, списка смежных вершин.

|  |  |
| --- | --- |
| 5 |  |

Матрица смежности - это квадратная матрица размера n x n, где n - число вершин в графе. Эта матрица используется для представления графа в виде таблицы, где каждый элемент матрицы a\_ij показывает, существует ли ребро между вершинами i и j.

Если в графе есть ребро между вершинами i и j, то a\_ij равно 1. Если ребра между этими вершинами нет, то a\_ij равно 0. Обратим внимание, что для неориентированного графа матрица смежности будет симметричной относительно главной диагонали.

**Матрица смежности:**

{0, 0, 1, 0, 0, 0, 0},

{0, 0, 0, 0, 0, 0, 0},

{0, 0, 0, 1, 0, 1, 0},

{0, 1, 0, 0, 1, 1, 1},

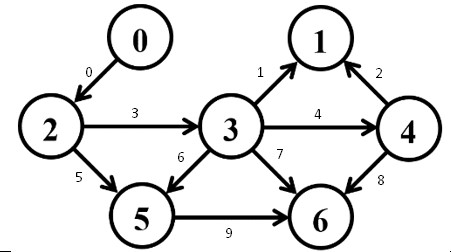
{0, 1, 0, 0, 0, 0, 1},

{0, 0, 0, 0, 0, 0, 1},

{0, 0, 0, 0, 0, 0, 0}



G = (V, E), V = {0, 1, 2, 3, 4, 5, 6}, E = {<0, 2>0, <1, 3>1, <1, 4>2, <2, 3>3, <3, 4>4, <2, 5>5, <3, 5>6, <3, 6>7, <4, 6>8, <5, 6>9}



\*если в двух словах, если ребро исходит от вершины, то 1, если входит, то -1, 0 – если ничего

**Матрица инцидентности:**

{1, 0, 0, 0, 0, 0, 0, 0, 0, 0},

{0, -1, -1, 0, 0, 0, 0, 0, 0, 0},

{-1, 0, 0, 1, 0, 1, 0, 0, 0, 0},

{0, 1, 0, -1, 1, 0, 1, 1, 0, 0},

{0, 0, 1, 0, -1, 0, 0, 0, 1, 0},

{0, 0, 0, 0, 0, -1, -1, 0, 0, 1},

{0, 0, 0, 0, 0, 0, 0, -1, -1, -1}

Список смежных вершин - это список всех вершин, с которыми заданная вершина в графе имеет ребра. Другими словами, это список вершин, которые непосредственно соединены с данной вершиной ребрами.

Например, если в графе есть вершина A, которая имеет ребра, соединяющие ее с вершинами B, C и D, то список смежных вершин для вершины A будет содержать вершины B, C и D.

Список смежных вершин может быть представлен в виде массива, списка или любой другой структуры данных, в которой хранятся вершины, с которыми данная вершина имеет ребра.

**Список смежных вершин:**

S0 = {2}

S1 = ∅

S2 = {3, 5}

S3 = {1, 4, 5, 6}

S4 = {1, 6}

S5 = {6}

S6 = ∅

***Задание 2.*** Осуществить алгоритмы поиска в ширину и глубину, а также алгоритма топологической сортировки аналогично примерам, рассмотренным на лекциях. Оформить отчет, включив в него **каждый** шаг выполнения алгоритмов.

**Алгоритм поиска в ширину (англ. breadth-first search, BFS)** позволяет найти кратчайшие пути из одной вершины невзвешенного графа до всех остальных вершин

**Поиск в ширину:**

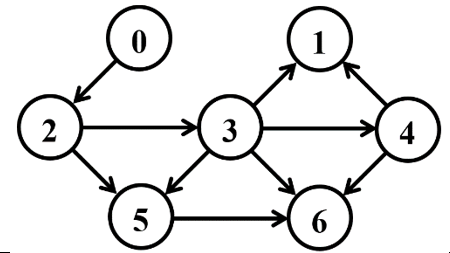
Q- для промежуточного хранения вершин(очередь)

массивы:

С- массив окраски вершин ( Б – не добавлена в очередь , С – добавлена в очередь , Ч – вышла из очереди )

D- массив расстояний

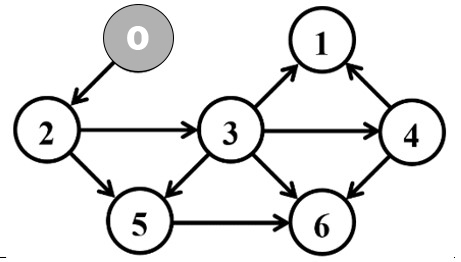
Р- массив предшествующих вершин



Шаг 1

Очередь: 0

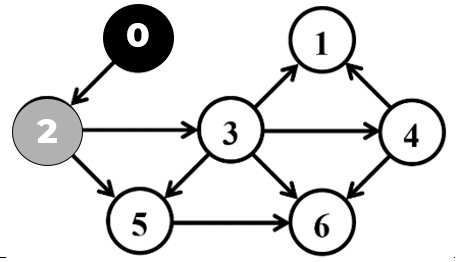
|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
| Окраска вершин | С | Б | Б | Б | Б | Б | Б |
| Расстояния | 0 | - | - | - | - | - | - |
| Массив предыдущих вершин | - | - | - | - | - | - | - |



Шаг 2

Очередь: 2

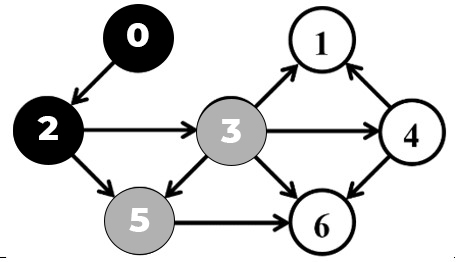
|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
| Окраска вершин | Ч | Б | С | Б | Б | Б | Б |
| Расстояния | 0 | - | 1 | - | - | - | - |
| Массив предыдущих вершин | - | - | 0 | - | - | - | - |



Шаг 3

Очередь: 3 5

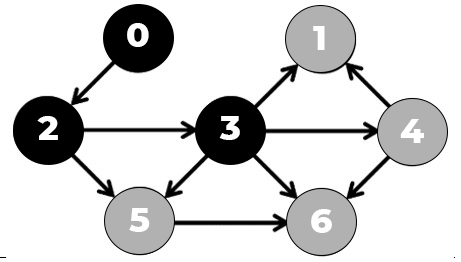
|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
| Окраска вершин | Ч | Б | Ч | С | Б | С | Б |
| Расстояния | 0 | - | 1 | 2 | - | 2 | - |
| Массив предыдущих вершин | - | - | 0 | 1 | - | 1 | - |



Шаг 4

Очередь: 5 1 4 6

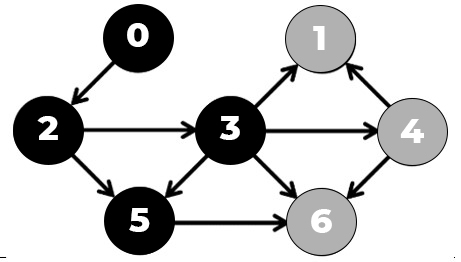
|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
| Окраска вершин | Ч | C | Ч | Ч | С | С | С |
| Расстояния | 0 | 3 | 1 | 2 | 3 | 2 | 3 |
| Массив предыдущих вершин | - | 2 | 0 | 1 | 2 | 1 | 2 |



Шаг 5

Очередь: 1 4 6

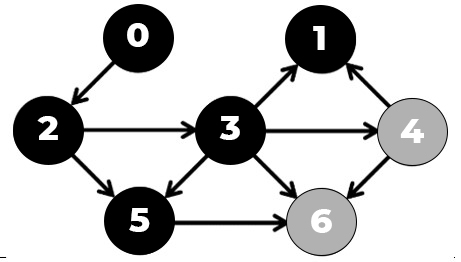
|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
| Окраска вершин | Ч | С | Ч | Ч | С | Ч | С |
| Расстояния | 0 | 3 | 1 | 2 | 3 | 2 | 3 |
| Массив предыдущих вершин | - | 2 | 0 | 1 | 2 | 1 | 2 |



Шаг 6

Очередь: 4 6

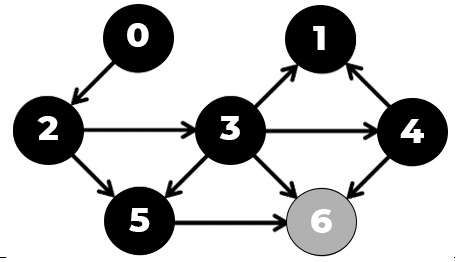
|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
| Окраска вершин | Ч | Ч | Ч | Ч | С | Ч | С |
| Расстояния | 0 | 3 | 1 | 2 | 3 | 2 | 3 |
| Массив предыдущих вершин | - | 2 | 0 | 1 | 2 | 1 | 2 |



Шаг 7

Очередь: 6

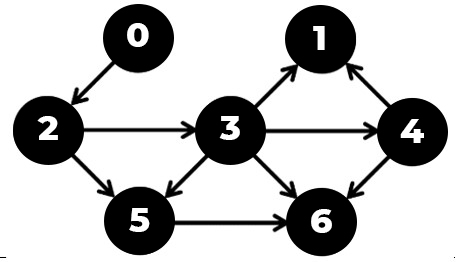
|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
| Окраска вершин | Ч | Ч | Ч | Ч | Ч | Ч | С |
| Расстояния | 0 | 3 | 1 | 2 | 3 | 2 | 3 |
| Массив предыдущих вершин | - | 2 | 0 | 1 | 2 | 1 | 2 |



Шаг 8

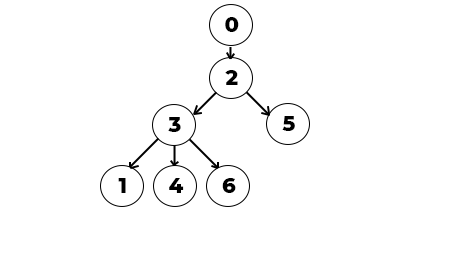
Очередь: …

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
| Окраска вершин | Ч | Ч | Ч | Ч | Ч | Ч | Ч |
| Расстояния | 0 | 3 | 1 | 2 | 3 | 2 | 3 |
| Массив предыдущих вершин | - | 2 | 0 | 1 | 2 | 1 | 2 |



Очередь пуста – следует конец обхода.

Результат:



**Алгоритм BFS** сводится к следующей последовательности шагов.

1. Инициализировать массивы **С**, **D**, **P**. Стартовую вершину **s** поместить в очередь **Q**. и окрасить в серый цвет: **C[s] = G**. Для стартовой вершины установить расстояние, равное нулю: **D[s] = 0**.
2. Если очередь **Q** пуста, то работа алгоритма завершена, в противном случае перейти к следующему шагу.
3. Выбрать из очереди **Q** вершину **k** и окрасить ее в черный цвет: **С[k] = B**.
4. Построить множества **J** вершин белого цвета смежных вершине **k**. Если таких вершин нет, то перейти к шагу 2, иначе – к следующему шагу.
5. Каждую вершину **j** из множества **J** поместить в очередь **Q**. Обычно (но не обязательно) в очередь вершины помещаются в порядке возрастания номеров.
6. Каждую вершину **j** из множества **J** окрасить в серый цвет: **С[j] = G**.
7. Для каждой вершины **j** из множества **J** вычислитьрасстояние: **D[j] = D[k] + 1**.
8. Для каждой вершины **j** из множества **J** указать предшествующую вершину: **P[j] = k**.
9. Перейти к шагу 3.

**Алгоритм поиска (или обхода) в глубину** (англ. depth-first search, DFS) позволяет построить обход графа, при котором посещаются все вершины, доступные из начальной вершины.

**Алгоритм поиска в глубину:**

T-шаг

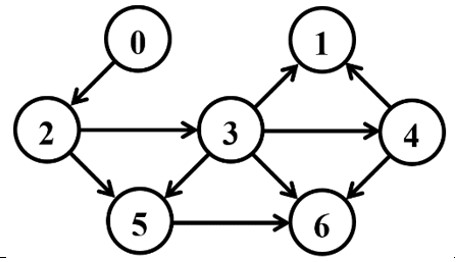
Стек посещённых вершин

Массив окраски вершин (Б – не посещена , С – в стеке , Ч – вышла из стека)

Массив расстояний

Массив предшествующих вершин

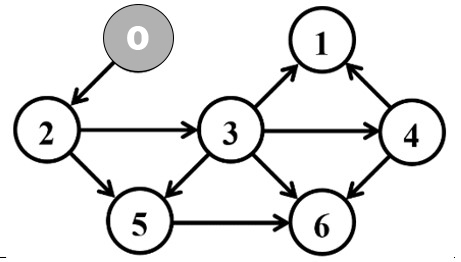
Массив шаг, на котором вершина окрашивается в черный цвет



Шаг 1

Стек:

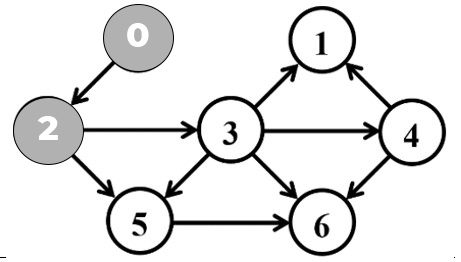
|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
| Окраска вершин | С | Б | Б | Б | Б | Б | Б |
| Расстояния | 0 |  |  |  |  |  |  |
| Номера предыдущих вершин |  |  |  |  |  |  |  |
| Номер шага , когда вершина выходит из алгоритма |  |  |  |  |  |  |  |



Шаг 2

Стек: 0

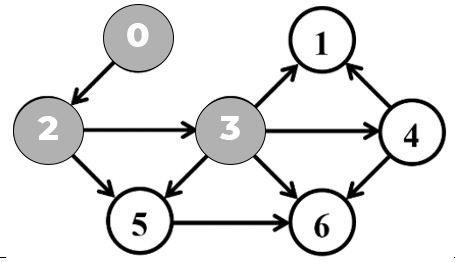
|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
| Окраска вершин | С | Б | С | Б | Б | Б | Б |
| Расстояния | 0 | - | 1 |  |  |  |  |
| Номера предыдущих вершин |  |  | 0 |  |  |  |  |
| Номер шага , когда вершина выходит из алгоритма |  |  |  |  |  |  |  |



Шаг 3

Стек: 0 2

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
| Окраска вершин | С | Б | С | С | Б | Б | Б |
| Расстояния | 0 | - | 1 | 2 |  |  |  |
| Номера предыдущих вершин |  |  | 0 | 2 |  |  |  |
| Номер шага , когда вершина выходит из алгоритма |  |  |  |  |  |  |  |

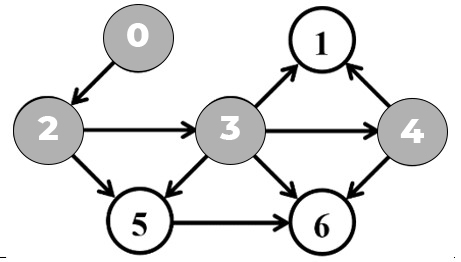


Шаг 4

Комментарий: выбираю 4, т.к. другого способа связать 4-ку нет

Стек: 0 2 3

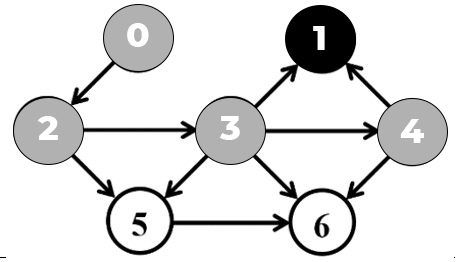
|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
| Окраска вершин | С | Б | С | С | С | Б | Б |
| Расстояния | 0 | - | 1 | 2 | 3 |  |  |
| Номера предыдущих вершин |  |  | 0 | 2 | 3 |  |  |
| Номер шага , когда вершина выходит из алгоритма |  |  |  |  |  |  |  |



Шаг 5

Стек: 0 2 3 4

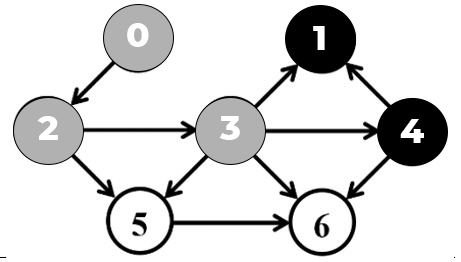
|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
| Окраска вершин | С | Ч | С | С | С | Б | Б |
| Расстояния | 0 | 4 | 1 | 2 | 3 |  |  |
| Номера предыдущих вершин |  | 4 | 0 | 2 | 3 |  |  |
| Номер шага , когда вершина выходит из алгоритма |  | 6 |  |  |  |  |  |



Шаг 6

Стек: 0 2 3

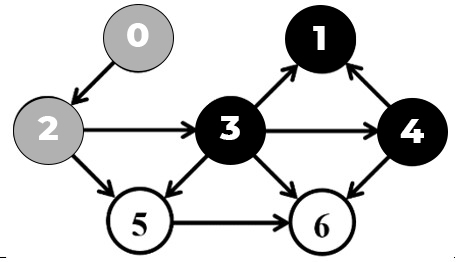
|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
| Окраска вершин | С | Ч | С | С | Ч | Б | Б |
| Расстояния | 0 | 4 | 1 | 2 | 3 |  |  |
| Номера предыдущих вершин |  | 4 | 0 | 2 | 3 |  |  |
| Номер шага , когда вершина выходит из алгоритма |  | 6 |  |  | 7 |  |  |



Шаг 7

Стек: 0 2

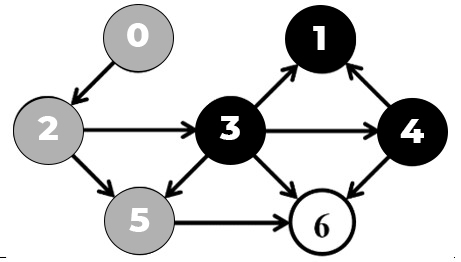
|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
| Окраска вершин | С | Ч | С | Ч | Ч | Б | Б |
| Расстояния | 0 | 4 | 1 | 2 | 3 |  |  |
| Номера предыдущих вершин |  | 4 | 0 | 2 | 3 |  |  |
| Номер шага , когда вершина выходит из алгоритма |  | 6 |  | 8 | 7 |  |  |



Шаг 8

Стек: 0 2

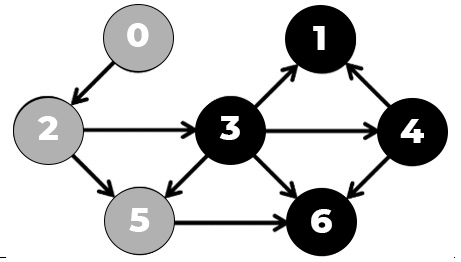
|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
| Окраска вершин | С | Ч | С | Ч | Ч | С | Б |
| Расстояния | 0 | 4 | 1 | 2 | 3 | 2 |  |
| Номера предыдущих вершин |  | 4 | 0 | 2 | 3 | 2 |  |
| Номер шага , когда вершина выходит из алгоритма |  | 6 |  | 8 | 7 |  |  |



Шаг 9

Стек: 0 2 5

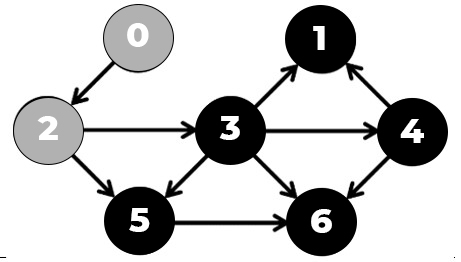
|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
| Окраска вершин | С | Ч | С | Ч | Ч | С | С |
| Расстояния | 0 | 4 | 1 | 2 | 3 | 2 | 3 |
| Номера предыдущих вершин |  | 4 | 0 | 2 | 3 | 2 | 5 |
| Номер шага , когда вершина выходит из алгоритма |  | 6 |  | 8 | 7 |  | 12 |



Шаг 10

Стек: 0 2

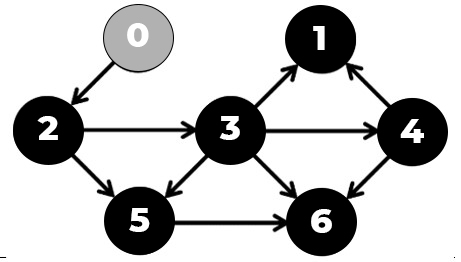
|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
| Окраска вершин | С | Ч | С | Ч | Ч | Ч | Ч |
| Расстояния | 0 | 4 | 1 | 2 | 3 | 2 | 3 |
| Номера предыдущих вершин |  | 4 | 0 | 2 | 3 | 2 | 5 |
| Номер шага , когда вершина выходит из алгоритма |  | 6 |  | 8 | 7 | 13 | 12 |



Шаг 11

Стек: 0

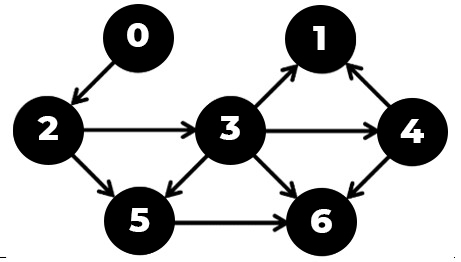
|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
| Окраска вершин | С | Ч | Ч | Ч | Ч | Ч | Ч |
| Расстояния | 0 | 4 | 1 | 2 | 3 | 2 | 3 |
| Номера предыдущих вершин |  | 4 | 0 | 2 | 3 | 2 | 5 |
| Номер шага , когда вершина выходит из алгоритма |  | 6 | 14 | 8 | 7 | 13 | 12 |



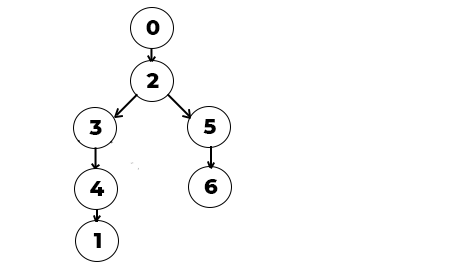
Шаг 12

Стек: …

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
| Окраска вершин | Ч | Ч | С | Ч | Ч | Ч | Ч |
| Расстояния | 0 | 4 | 1 | 2 | 3 | 2 | 3 |
| Номера предыдущих вершин |  | 4 | 0 | 2 | 3 | 2 | 5 |
| Номер шага , когда вершина выходит из алгоритма | 15 | 6 | 14 | 8 | 7 | 13 | 12 |



Результат:



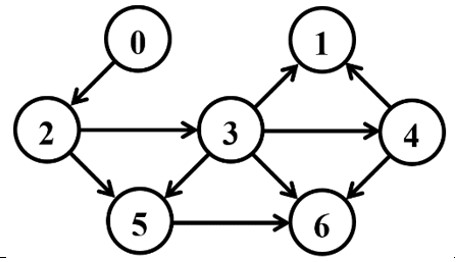
В основе алгоритма DFS лежит рекурсивная процедура **Visit**, имеющая один входной параметр **k** – вершину графа.

Опишем пошагово процедуру **Visit**.

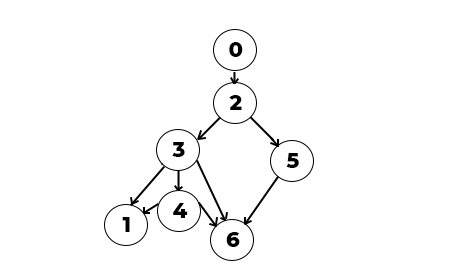
1. Принять параметр **k** – вершину графа.
2. Вершину **k**  окрасить в серый цвет: **C[k] = G**.
3. Увеличить номер шага: **t = t + 1**.
4. Подсчитать расстояние до вершины: **D[k] = t**. Расстояние до вершины в алгоритме DFS совпадает с номером шага, на котором эта вершина была обнаружена (окрашена в серый цвет).
5. Построить множества **J** вершин белого цвета, смежных вершине **k**. Если таких вершин нет, то перейти к шагу 8.
6. Для каждой вершины **j** из множества **J** указать предшествующую вершину: **P[j] = k**.
7. Для каждой вершины **j** из множества **J** выполнить процедуру **Visit**.
8. Вершину **k**  окрасить в черный цвет: **C[k] = B**.
9. Увеличить номер шага: **t = t + 1**.
10. Отметить время фиксации вершины: **F[k] = t**.

**Топологическая сортировка**

**Топологическая сортировка −** это процедура упорядочивания вершин бесконтурного ориентированного графа, не имеющего циклов (ациклического графа). В результате топологической сортировки для вершин графа определяется такой порядок, что если их расположить на рисунке в соответствии с этим порядком сверху вниз, то дуги будут направлены только от верхних вершин к нижним**.**



**Результат**



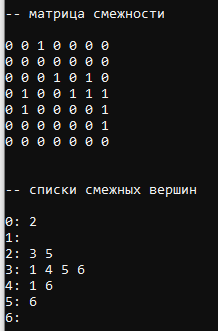
***Задание 3.*** Осуществить программную реализацию алгоритмов на C++. Разработать структуры **AMatrix** и **АList**  для представления ориентированного графа матричным и списковым способом. Разработать функции преобразования из одного способа представления в другой. Разработать функцию **BFS** обхода вершин графа, используя метод поиска в ширину. Продемонстрировать работу функции. Копии экрана вставить в отчет.

|  |
| --- |
| // ---BFS.h  //  #pragma once  #include "Graph.h"  #include <queue>  struct BFS // breadth-first search поиск в ширину (связный граф)  {  const static int INF = 0x7fffffff;  const static int NIL = -1;  enum Color { WHITE, GRAY, BLACK }; //  const graph::AList\* al; // исходный граф  Color\* c; // цвет вершины  int\* d; // расстояние до вершины  int\* p; // предшествующая вершина  std::queue<int> q; // очередь  BFS(const graph::AList& al, int s);  BFS(const graph::AMatrix& am, int s);  void init(const graph::AList& al, int s);  int get(); // получить следующую вершину  }; |

Листинг 3.1 — файл BFS.h

|  |
| --- |
| // ---BFS.cpp  //  #include "BFS.h"  void BFS::init(const graph::AList& al, int s)  {  this->al = &al;  this->c = new Color[this->al->n\_vertex];  this->d = new int[this->al->n\_vertex];  this->p = new int[this->al->n\_vertex];  for (int i = 0; i < this->al->n\_vertex; i++)  {  this->c[i] = WHITE;  this->d[i] = INF;  this->p[i] = NIL;  };  this->c[s] = GRAY;  this->q.push(s);  };  BFS::BFS(const graph::AList& al, int s) { this->init(al, s); };  BFS::BFS(const graph::AMatrix& am, int s)  {  this->init(\*(new graph::AList(am)), s);  };  int BFS::get()  {  int rc = NIL, v = NIL;  if (!this->q.empty())  {  rc = this->q.front();  for (int j = 0; j < this->al->size(rc); j++)  if (this->c[v = this->al->get(rc, j)] == WHITE)  {  this->c[v] = GRAY;  this->d[v] = this->d[rc] + 1;  this->p[v] = rc;  this->q.push(v);  };  this->q.pop();  this->c[rc] = BLACK;  };  return rc;  } |

Листинг 3.2 — файл BFS.cpp



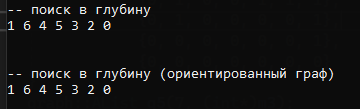
***Задание 4.*** Разработать функцию **DFS**  обхода вершин графа, используя метод поиска глубину. Продемонстрировать работу функции. Копии экрана вставить в отчет.

|  |
| --- |
| // ---DFS.h  //  #pragma once  #include "Graph.h"  #include <vector>  struct DFS // depth-first search поиск в глубину  {  const static int NIL = -1;  enum Color { WHITE, GRAY, BLACK }; //  const graph::AList\* al; // исходный граф  Color\* c; // цвет вершины  int\* d; // время обнаружения  int\* f; // время завершения обработки  int\* p; // предшествующая вершина  int t; // текущее время  DFS(const graph::AList& al);  DFS(const graph::AMatrix& am);  std::vector <int> topological\_sort; //результат топологической сортировки  void visit(int v);  void init(const graph::AList& al);  int get(int i); // получить вершину  }; |

Листинг 4.1 — файл DFS.h

|  |
| --- |
| // ---DFS.cpp  //  #include "DFS.h"  #define NINF 0x80000000  #define INF 0x7fffffff  void DFS::init(const graph::AList& al)  {  this->al = &al;  this->c = new Color[this->al->n\_vertex];  this->d = new int[this->al->n\_vertex];  this->f = new int[this->al->n\_vertex];  this->p = new int[this->al->n\_vertex];  this->t = 0;  for (int i = 0; i < this->al->n\_vertex; i++)  {  this->c[i] = WHITE;  this->d[i] = this->f[i] = 0;  this->p[i] = NIL;  };  for (int i = 0; i < this->al->n\_vertex; i++)  if (this->c[i] == WHITE)  {  this->visit(i);  this->topological\_sort.push\_back(i);  }  };  DFS::DFS(const graph::AList& al) { this->init(al); };  DFS::DFS(const graph::AMatrix& am)  {  this->init(\*(new graph::AList(am)));  };  void DFS::visit(int u)  {  int v = NIL;  this->c[u] = GRAY;  this->d[u] = ++(this->t);  for (int j = 0; j < this->al->size(u); j++)  if (this->c[v = this->al->get(u, j)] == WHITE)  {  this->p[v] = u;  this->visit(v);  this->topological\_sort.push\_back(v);  }  this->c[u] = BLACK;  this->f[u] = ++(this->t);  };  int DFS::get(int i)  {  int j = 0, min1 = INF, min2 = NINF, ntx = NIL;  for (int j = 0; j <= i; j++) // iая статистика  {  for (int k = 0; k < this->al->n\_vertex; k++)  if (this->f[k] < min1 && this->f[k] > min2)  {  min1 = this->f[k]; ntx = k;  };  min2 = min1; min1 = INF;  };  return ntx;  }; |

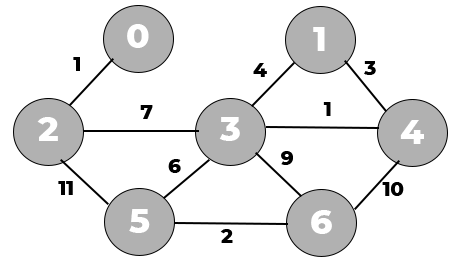
Листинг 4.2 — файл DFS.cpp



***Задание 5.*** Доработайте функцию **DFS**,для выполнения топологической сортировки графа. Продемонстрировать работу функции. Копии экрана вставить в отчет.

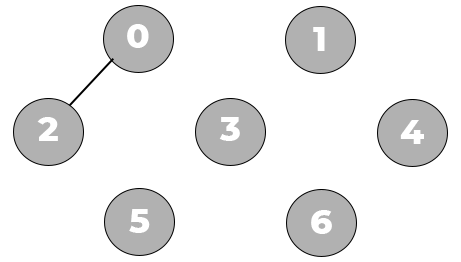


***Задание 6.*** По графу, соответствующему варианту составить минимальное остовное дерево по алгоритму Прима. Шаги построения отразить в отчете.

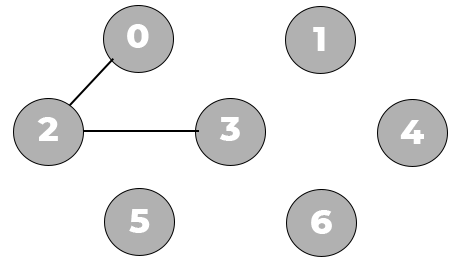


Алгоритм работает путем построения дерева по одной вершине за раз, начиная с произвольной начальной вершины, на каждом шаге добавляя максимально дешевое соединение из дерева в другую вершину

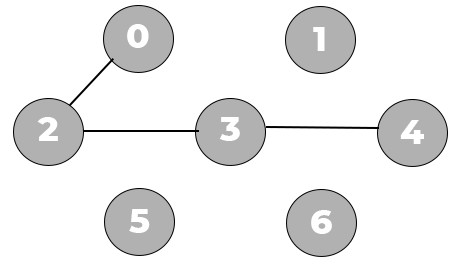
***Шаг 1:***



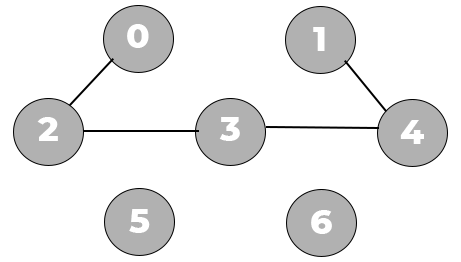
***Шаг 2:***



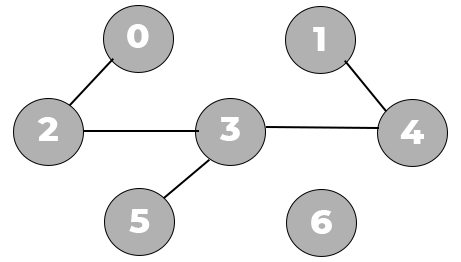
***Шаг 3:***



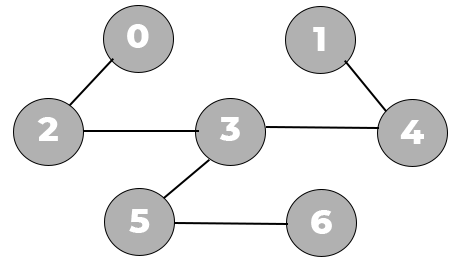
***Шаг 4:***



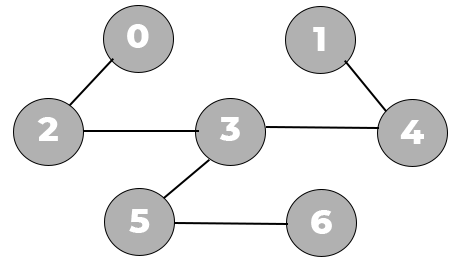
***Шаг 5:***



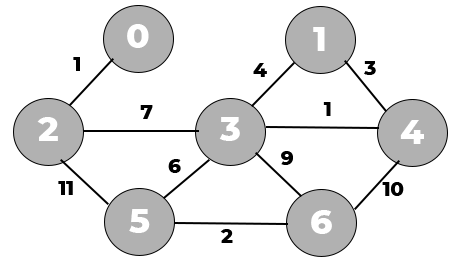
***Шаг 6:***



***Результат:***

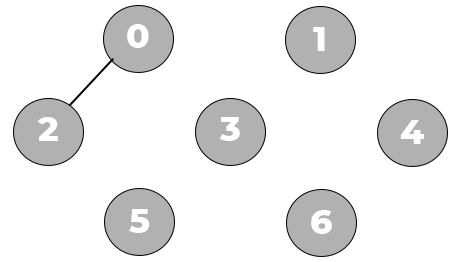


***Задание 7.*** По графу, соответствующему варианту составить минимальное остовное дерево по алгоритму Крускала. Шаги построения отразить в отчете.

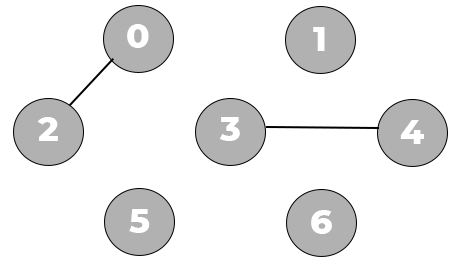


В начале текущее множество рёбер устанавливается пустым. Затем, пока это возможно, проводится следующая операция: из всех рёбер, добавление которых к уже имеющемуся множеству не вызовет появление в нём цикла, выбирается ребро минимального веса и добавляется к уже имеющемуся множеству. Когда таких рёбер больше нет, алгоритм завершён.

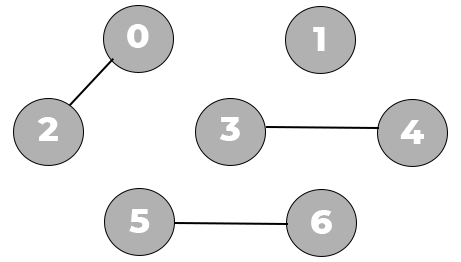
***Шаг 1:***

******

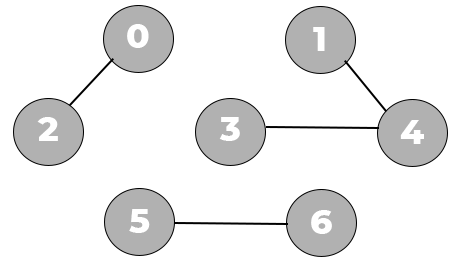
***Шаг 2:***

******

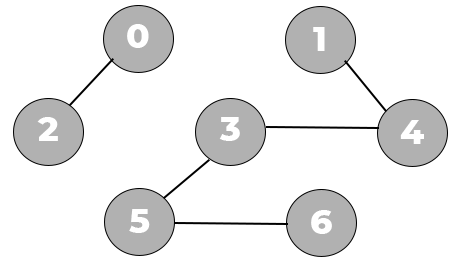
***Шаг 3:***

******

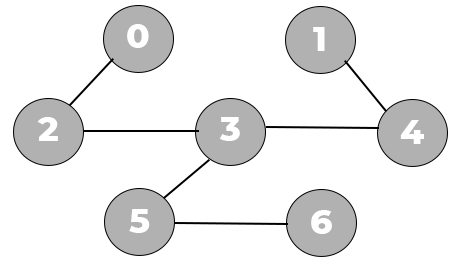
***Шаг 4:***



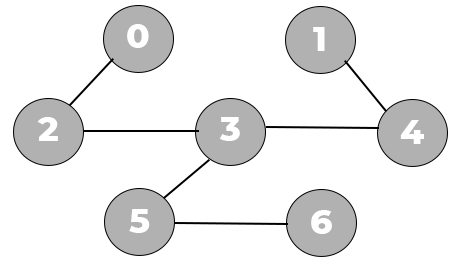
***Шаг 5:***



***Шаг 6:***



***Результат:***

******

***Вывод:*** в ходе лабораторной работы я освоила сущность и программную реализацию: а) способов представления графов; б) алгоритмов поиска в ширину и глубину; в) алгоритма топологической сортировки графов и разобрала алгоритм Прима и алгоритм Крускала