|  |  |
| --- | --- |
| |  | | --- | | МИНОБРНАУКИ РОССИИ  Федеральное государственное  бюджетное образовательное учреждение  высшего образования  **«МИРЭА – Российский технологический университет»**  **РТУ МИРЭА** | |

Институт Информационных технологий

Кафедра корпоративных информационных систем

**КУРСОВАЯ РАБОТА**

по дисциплине «Структуры и алгоритмы обработки данных»

**Тема курсовой работы** Сортировка ориентированных ацикличных графов с помощью алгоритмов топологической сортировки

**Студент группы** ИКБО-12-17 Лисовой А.А. \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

*(подпись студента)*

**Руководитель курсовой работы** Адрианова Е.Г.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

*(подпись руководителя)*

Работа представлена к защите «20» декабря 2018 г.

Допущен к защите «\_\_\_\_\_\_»\_\_\_\_\_\_\_\_\_\_\_\_\_2018 г.

Москва, 2018

|  |  |
| --- | --- |
| |  | | --- | | МИНОБРНАУКИ РОССИИ  Федеральное государственное  бюджетное образовательное учреждение  высшего образования  **«МИРЭА – Российский технологический университет»**  **РТУ МИРЭА** | |

Институт Информационных технологий

Кафедра корпоративных информационных систем

Утверждаю

Заведующий

кафедрой Корпоративных

информационных систем

проф. Петров А.Б.

Подпись \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

«\_\_\_\_\_\_» \_\_\_\_\_\_\_\_\_\_\_\_2018 г.

**ЗАДАНИЕ**

**на выполнение курсовой работы** по дисциплине

«Структуры и алгоритмы обработки данных»

Студент Лисовой Андрей Анатольевич Группа ИКБО-12-17

**Тема** Сортировка ориентированных ацикличных графов с помощью алгоритмов топологической сортировки

**Исходные данные:** техническое задание, стандарт PEP8, документация языка Python, литература по теории графов

**Перечень вопросов, подлежащих разработке, и обязательного графического материала:**

1.Изучение существующих алгоритмов топологической сортировки и принципов их работы

2.Реализация алгоритмов топологической сортировки на языке Python

3.Реализация приложения, демонстрирующего функционирование разработанных алгоритмов топологической сортировки

4.Отчет по курсовой работе в виде расчетно-пояснительной записки .

**Срок представления к защите курсовой работы:** до «20» декабря 2018 г.

**Задание на курсовую работу выдал** \_\_\_\_\_\_\_\_\_\_\_\_ (Андрианова Е.Г.)

*(подпись руководителя)*

«\_\_\_»\_\_\_\_\_\_\_\_\_\_\_ 2018 г.

**Задание на курсовую работу получил**  \_\_\_\_\_\_\_\_\_\_\_\_ (Лисовой А.А.)

*(подпись студента)*

Оглавление

[ВВЕДЕНИЕ 4](#_Toc532765514)

[ОСНОВНАЯ ЧАСТЬ 5](#_Toc532765515)

[1. ВВЕДЕНИЕ В ТОПОЛОГИЧЕСКУЮ СОРТИРОВКУ 5](#_Toc532765516)

[2. АЛГОРИТМ ОБХОДА В ГЛУБИНУ 9](#_Toc532765517)

[2.1 Описание алгоритма 9](#_Toc532765518)

[2.2 Кодирование алгоритма 11](#_Toc532765519)

[2.3 Тестирование алгоритма 11](#_Toc532765520)

[3. АЛГОРИТМ КАНА 13](#_Toc532765521)

[3.1 Описание алгоритма 13](#_Toc532765522)

[3.2 Кодирование алгоритма 15](#_Toc532765523)

[3.3 Тестирование алгоритма 15](#_Toc532765524)

[4. АЛГОРИТМ ТАРЬЯНА 18](#_Toc532765525)

[4.1 Описание алгоритма 18](#_Toc532765526)

[4.2 Кодирование алгоритма 19](#_Toc532765527)

[4.3 Тестирование алгоритма 20](#_Toc532765528)

[5 ОПИСАНИЕ ИНТЕРФЕЙСА ПРОГРАММЫ 22](#_Toc532765529)

[5.1 Ввод 22](#_Toc532765530)

[5.2 Вывод 22](#_Toc532765531)

[5.3 Организация работы сортировок 23](#_Toc532765532)

[ЗАКЛЮЧЕНИЕ 24](#_Toc532765533)

[СПИСОК ИСПОЛЬЗОВАННОЙ ЛИТЕРАТУРЫ 25](#_Toc532765534)

[ПРИЛОЖЕНИЯ 26](#_Toc532765535)

[1. ИСХОДНЫЙ КОД ПРОГРАММЫ 26](#_Toc532765536)

[Node.py 26](#_Toc532765537)

[LQS.py 26](#_Toc532765538)

[List.py……………………………………………………………………………………………………………. 26](#_Toc532765539)

[StackList.py 29](#_Toc532765540)

[QueueList.py 30](#_Toc532765541)

[Graph.py 30](#_Toc532765542)

[Interface.py 34](#_Toc532765543)

[Main.py 35](#_Toc532765544)

[2. ПРЕЗЕНТАЦИЯ 36](#_Toc532765545)

# ВВЕДЕНИЕ

Топологическая сортировка – это математическая операция из области теории графов, процесс поиска такого линейного порядка на вершинах графа, чтобы любое его ребро вело от вершины с меньшим порядковым номером к вершине с большим порядковым номером.

Сегодня, топологическая сортировка, как один из разделов теории графов, находит множественное применение, особенно в таких дисциплинах, как математическая логика, дискретная математика, информатика и современная физика. Практическое применение данной сортировки можно наблюдать при построении алгоритмов действий, где всякое из действий может строго зависеть от другого, например: организация движения поездов в метрополитене, обеспечение корректного графика движения самолетов в аэропорту, установки программ при помощи пакетного менеджера, создания карты сайта, сборки исходных кодов программ при помощи makefile (англ. Исполняемый файл), а так же для распараллеливания уже существующих алгоритмов и определения порядка отрисовки объектов в изометрической графике.

Иными словами, топологическая сортировка – достаточно часто применяемая математическая операция, подчиняющаяся строгому набору условий и правил. Следовательно, её можно автоматизировать с помощью программного приложения, что и было успешно сделано в данной курсовой работе.

# ОСНОВНАЯ ЧАСТЬ

## 1. ВВЕДЕНИЕ В ТОПОЛОГИЧЕСКУЮ СОРТИРОВКУ

Цель данной работы – создание программного приложения в образовательных целях для введения в такой раздел теории графов, как топологическая сортировка и наглядной демонстрации наиболее известных и эффективных алгоритмов её реализации.

Известно, что топологическая сортировка возможна только для ориентированного ациклического графа. Ориентированный ациклический граф (англ. Directed Acyclic Graph) – это такой ориентированный (направленный) граф, что не содержит в своей структуре контуров (циклов). Простейший пример такого графа – ориентированное дерево.

Степень вершины графа – это количество ребер, связывающих вершину графа с соседними. Степень вершины может быть входящая и исходящая (для неориентированных графов входящая степень равна исходящей):

1. Входящая степень вершины v это количество ребер вида

(i, v), то есть количество ребер которые «входят» в v.

1. Исходящая степень вершины v это количество ребер вида

(v , i), то есть количество ребер которые «выходят» из v.

Путь - это конечная последовательность вершин графа, в которой каждые две вершины идущие подряд соединены ребром. Путь может быть ориентированным или неориентированным в зависимости от графа.

Существует два способа представления графа, в виде списков смежности (Подходит для разреженных графов, то есть графов у которых количество рёбер гораздо меньше чем количество вершин в квадрате) и в виде матрицы смежности (Этот способ является удобным для представления плотных графов, в которых количество рёбер примерно равно количеству вершин в квадрате). Оба способа подходят для представления ориентированных и неориентированных графов.

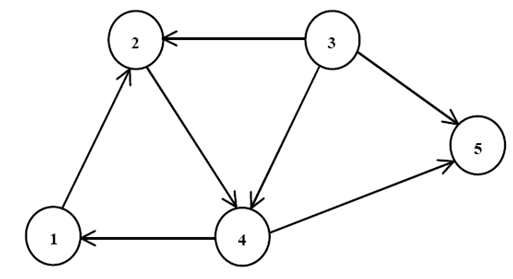


Рис. 1 – Граф, содержащий в себе цикл. Для него не применим алгоритм стандартной топологической сортировки.

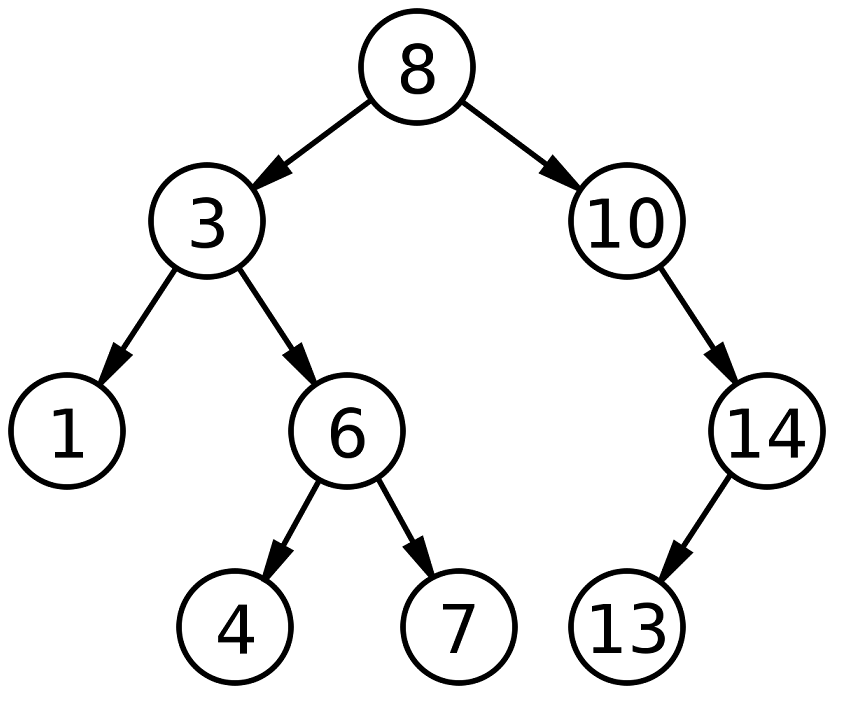


Рис. 2 – Ориентированное дерево – простейший пример направленного ациклического графа, к которому применима топологическая сортировка.

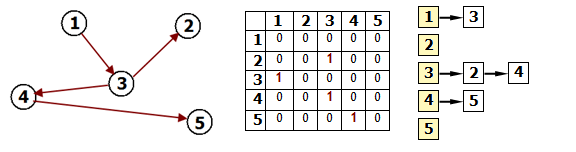


Рис. 3 - Матрица инцидентности и список смежности для ориентированного ациклического графа.

С математической точки зрения, цель топологической сортировки – преобразовать частичный порядок вершин графа в линейный. Множество M называется частично упорядоченным, если над его элементами определено отношение, которое мы назовем "x предшествует y" и обозначим x << y, удовлетворяющее следующим свойствам для любых элементов x, y и z из M:

1. Не x << x (Антирефлексивность)
2. Если x << y, то не y << x (Антисимметричность)
3. Если x << y и y << z, то x << z (Транзитивность).

Множество может быть частично упорядоченным только при условии его конечности.

Частичное упорядочение на конечном множестве всегда можно изобразить с помощью диаграммы вершин некоторого графа, в которой элементы представляются вершинами графа, а отношения представляются дугами между этими вершинами; x << y означает, что от вершины, помеченной x, к вершине y существует путь, идущий вдоль дуг в соответствии с их направлением. Свойство частичного упорядочения означает, что в диаграмме графа нет замкнутых контуров - циклов.

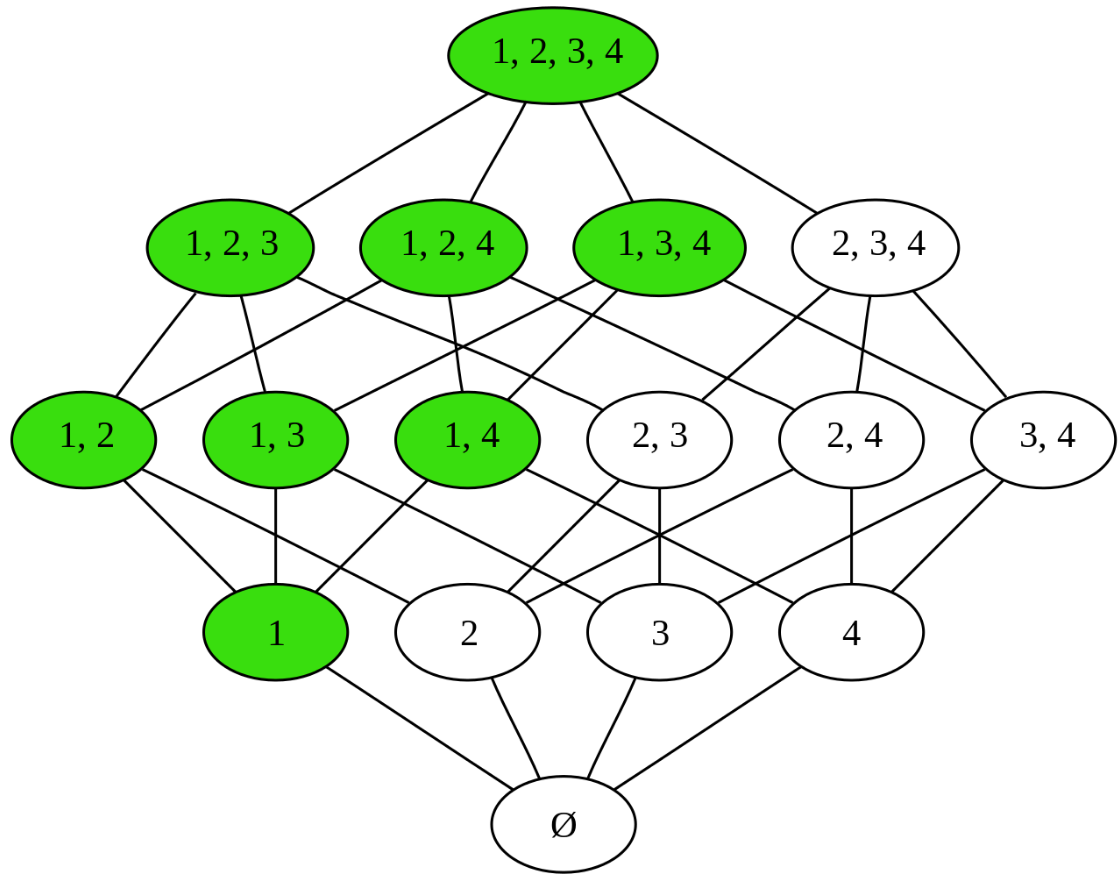


Рис. 4 – Пример частично упорядоченного множества, в данном случае относительно вершины с значением единицы.

Главная задача топологической сортировки состоит в том, чтобы перевести частичное упорядочение в линейное упорядочение, а значит расположить элементы в такую последовательность x1, x2, ..., xn, что если

xj << xk, то j < k. Существование такого расположения не является заведомо очевидным, однако оно абсолютно невозможно, если имеется хотя бы один цикл.

Чтобы найти хотя бы одно из возможных линейных упорядочений, нужно начать с того, чтобы выбрать одну из вершин, которой не предшествует никакая другая, после чего исключить данную вершину из множества. Множество по прежнему остается частично упорядоченным, позволяя рекурсивно повторять данный алгоритм, пока оно не станет пустым.

## 2. АЛГОРИТМ ОБХОДА В ГЛУБИНУ

### 2.1 Описание алгоритма

Поиск в глубину или обход в глубину (англ. Depth-first search, сокращенно DFS) — один из основных и самых известных методов обхода графа. Алгоритм поиска описывается следующим образом: для каждой не пройденной вершины необходимо найти все не пройденные смежные вершины и рекурсивно повторить поиск для них.

Более подробно данный алгоритм можно описать следующим образом: Выбираем вершину, входящая степень которой равна нулю, после чего начинаем обход в глубину, и после того, как вершина обработана, заносим ее в стек. По окончании обхода в глубину вершины достаются из стека. Новые номера присваиваются в порядке вытаскивания из стека. Во время обхода в глубину для обозначения состояния вершин используется 3 цвета. Изначально все вершины белые. В тот момент, когда мы впервые посещаем вершину, красим ее в серый цвет. Когда просмотрен список всех смежных с ней вершин, то есть, из неё больше невозможно никуда пройти дальше, красим ее в черный цвет.

Формально данный алгоритм можно представить следующим образом:

Помечаем все вершины графа, как не пройденные, после чего выбираем любую вершину из еще не пройденных, после чего рекурсивно ищем все исходящие из нее вершины, помечая их как пройденные - серым цветом. Для каждой из этих смежных выбранной вершин так же ищем смежные и помечаем их пройденными, после чего заносим изначально выбранную вершину в список отсортированных вершин, помечая её черным цветом. Повторяем шаги 1 и 2, пока все вершины не окажутся пройденными.

Серьезным недостатком данного алгоритма является то, что на очень больших графах множественный рекурсивный вызов функции обхода очень сильно нагружает стек вызовов, провоцируя риск возникновения переполнения стека, что может привести к аварийному завершению работы программы.

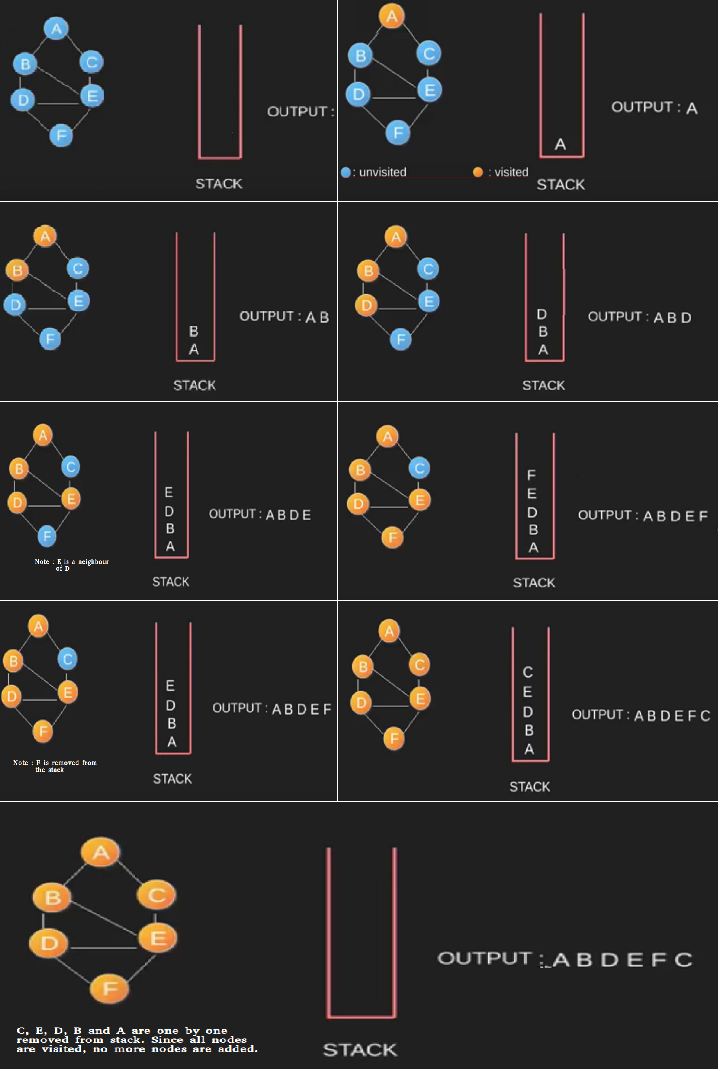


Рис. 5 – Графическая демонстрация метода обхода в глубину.

### 2.2 Кодирование алгоритма

1. # Топологическая сортировка обходом в глубину
2. **def** Deep\_First\_Topological\_Sort(self):
3. # Проверка на введённость графа
4. **if** self.\_edges\_value > 0:
5. # Граф
6. Graph = self.\_edges
7. # Дополним конечные вершины графа
8. BufferList = List()
9. **for** Vertex **in** Graph:
10. **for** Element **in** Graph[Vertex]:
11. **if** Element **not** **in** Graph.keys() **and** Element **not** **in** BufferList:
12. BufferList.Append(Element)
13. **for** Element **in** BufferList:
14. Graph[Element] = List()
15. # Результирующий список
16. Result = List()
17. # Словарь "цветов" вершин
18. Colors = { Vertex: "White" **for** Vertex **in** Graph}
19. # Цикл обхода вершин графа
20. **for** Vertex **in** Graph:
21. # Если вершина не посещена, она будет белого цвета
22. **if** Colors[Vertex] == "White":
23. self.\_Deep\_First\_Sort\_Recursion(Graph, Vertex, Colors, Result)
24. **if** Result.length != Graph.\_\_len\_\_():
25. **print**("Сортировка невозможна, так как исходный граф содержит цикл")
26. **print**("")
27. **return**
28. Result.Reverse()
29. **print**("Топологически отсортированный граф: ")
30. **print**(Result)
31. **print**("")
32. **else**:
33. **print**("Ошибка: граф не был введен")
34. **print**("")
36. # Рекурсивный цикл обхода вершин для DFS
37. **def** \_Deep\_First\_Sort\_Recursion(self, Graph, Vertex, Colors, Result):
38. # Обозначим посещенную вершину серым цветом
39. Colors[Vertex] = "Gray"
40. # Цикл обхода преемников посещенной вершины
41. **for** SubVertex **in** Graph[Vertex]:
42. # Если из посещенной вершины попали в посещенную - обнаружен цикл
43. **if** Colors[SubVertex] == "Gray":
44. Founded\_Cycle = True
45. **return**
46. # Если вершина-преемник не посещена, то рекурсивно обойдем и ее
47. **if** Colors[SubVertex] == "White":
48. self.\_Deep\_First\_Sort\_Recursion(Graph, SubVertex, Colors, Result)
49. # Когда дальнейший обход невозможен, покрасим вершину в черный цвет
50. # и добавим в результирующий список
51. Colors[Vertex] = "Black"
52. Result.Append(Vertex)

### 2.3 Тестирование алгоритма

Протестируем представленный выше алгоритм для графа, что был изображен на рис. 5, представив исходный граф, как множество(словарь) его вершин:

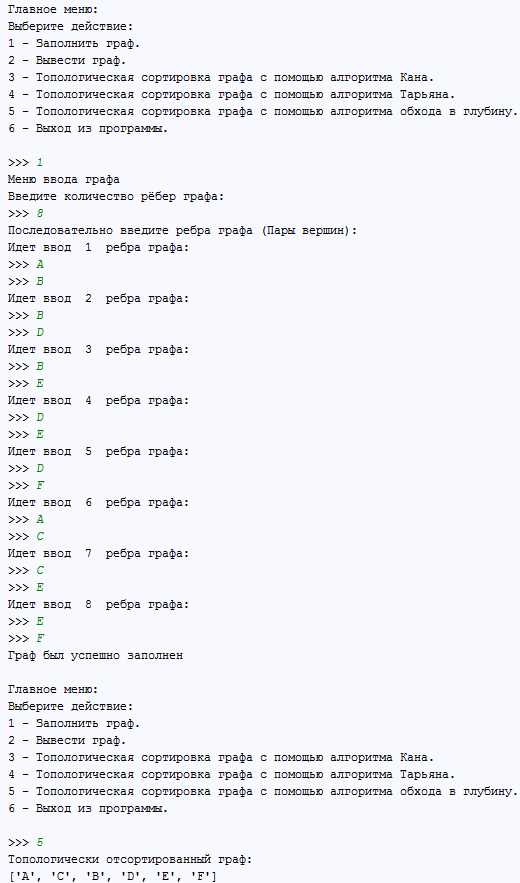


Рис. 6 Тестирование алгоритма обхода в глубину.

## 3. АЛГОРИТМ КАНА

### 3.1 Описание алгоритма

Алгоритм Кана был создан в 1962 году и является одним из старейших из известных алгоритмов топологической сортировки. Он является наиболее приспособленным для исполнения вручную. Данный алгоритм топологической сортировки применим только для направленных ациклических графов. Суть алгоритма заключается в последовательной сортировке вершин графа по критерию уровня их входящей степени (количества ребер, входящих в данную вершину) и записи их в список.

Более подробно данный алгоритм можно описать как процесс последовательной записи всех вершин графа в очередь, опираясь на их входящую степень, причем таким образом, что первыми в очереди будут находится элементы с нулевой входящей степенью, а далее в очередь будут помещаться все исходящие вершины от вершин с нулевой входящей степенью. При удалении вершины с нулевой входящей степенью из очереди, входящие степени исходящих из нее вершин должны понижаться на единицу.

Формально данный алгоритм можно представить следующим образом:

1. Записать все вершины графа в ассоциативный контейнер, например, словарь, подсчитав входящую степень каждой.
2. Выбрать все вершины графа с нулевой входящей степенью, после чего поместить их в очередь.
3. Удалить вершину из очереди, записав её в результирующий список. После чего инкрементировать количество посещенных узлов, уменьшить входящую степень всем соседним вершинам, и, если их входящая степень стала нулевой – добавить их в очередь.
4. Рекурсивно повторять третий шаг, пока очередь не опустеет окончательно
5. Проверить количество вершин в результирующем списке: если оно совпадает с количеством вершин графа, следует его вывести, иначе выдать сообщение об ошибке, так как исходный граф содержит контур(цикл), что недопустимо для данного алгоритма.

Одним из главных достоинств данного алгоритма является отсутствие рекурсивного вызова функций, что позволяет проводить сортировку графов любого размера без возникновения риска переполнения стека. Так же следует помнить, что отсортированная последовательность вершин графа прямо зависит от того, какая из вершин с нулевой входящей степенью первой была помещена в очередь.

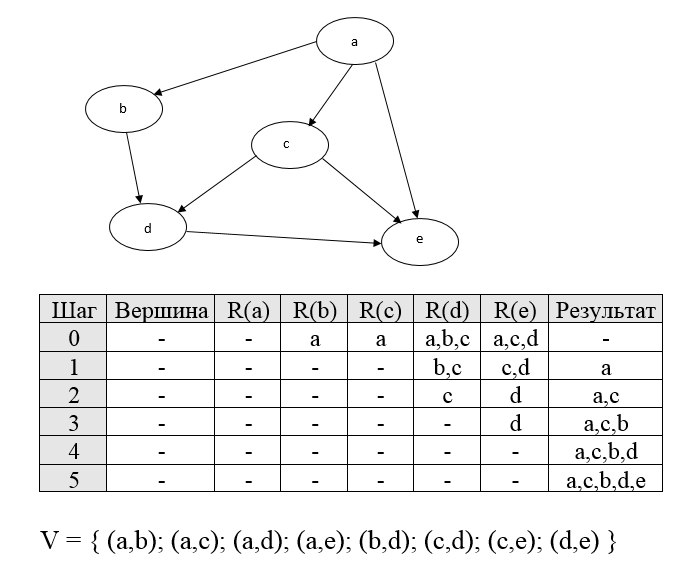


Рис. 7 – Пример топологической сортировки с помощью алгоритма Кана: при сортировке вместо вершины c может быть выбрана вершина b, так как между ними не был задан порядок.

### 3.2 Кодирование алгоритма

1. # Топологическая сортировка с помощью алгоритма Кана
2. **def** Kahn\_Topological\_Sort(self):
3. # Проверка на введённость графа
4. **if** self.\_edges\_value > 0:
5. # Копия графа
6. Graph = self.\_edges
7. # Список с отсортированным графом
8. Topological\_Sorted\_List = List()
9. # Очередь вершин с нулем входящих связей
10. Zero\_In\_Degree\_Vertex\_Queue = QueueList()
11. # Словарь входящих связей соседних вершин
12. In\_Degree = {Key: 0 **for** Key **in** Graph}
14. # Шаг 1: Обойти граф и посчитать связи соседних вершин
15. **for** Key **in** Graph:
16. **for** Value **in** Graph[Key]:
17. # Если вершина не была обозначена ранее, добавим её
18. **if** Value **in** In\_Degree:
19. In\_Degree[Value] += 1
20. **else**:
21. In\_Degree[Value] = 1
23. # Шаг 2: Обновим изначальный граф
24. **for** Key **in** In\_Degree:
25. **if** Key **in** In\_Degree **and** Key **not** **in** Graph:
26. Graph[Key] = List()
28. # Шаг 3: Поиск вершин с нулем входящих связей
29. **for** Key **in** In\_Degree:
30. **if** In\_Degree[Key] == 0:
31. Zero\_In\_Degree\_Vertex\_Queue.Enqueue(Key)
33. # Шаг 4: Обработка узлов с нулем входящих связей
34. **while** Zero\_In\_Degree\_Vertex\_Queue.length > 0:
35. Value = Zero\_In\_Degree\_Vertex\_Queue.Dequeue()
36. Topological\_Sorted\_List.Append(Value)
38. # Шаг 5: Обновим словарь входящих связей
39. **for** Neighbour **in** Graph[Value]:
40. In\_Degree[Neighbour] -= 1
41. **if** (In\_Degree[Neighbour] == 0):
42. Zero\_In\_Degree\_Vertex\_Queue.Enqueue(Neighbour)
44. # Шаг 6: Проверка на циклы и вывод отсортированного графа
45. **if** Topological\_Sorted\_List.length == len(Graph):
46. **print**("Отсортированный с помощью алгоритма Кана граф: ")
47. **print**(Topological\_Sorted\_List)
48. **print**("")
49. **else**:
50. **print**("Ошибка: исходный граф содержит циклы")
51. **print**("")
52. **else**:
53. **print**("Ошибка: граф не был введен")
54. **print**("")

### 3.3 Тестирование алгоритма

Для начала протестируем представленный алгоритм на графе, изображенном на рис. 7, так как он не содержит в себе циклов и предполагает стандартную работу алгоритма:

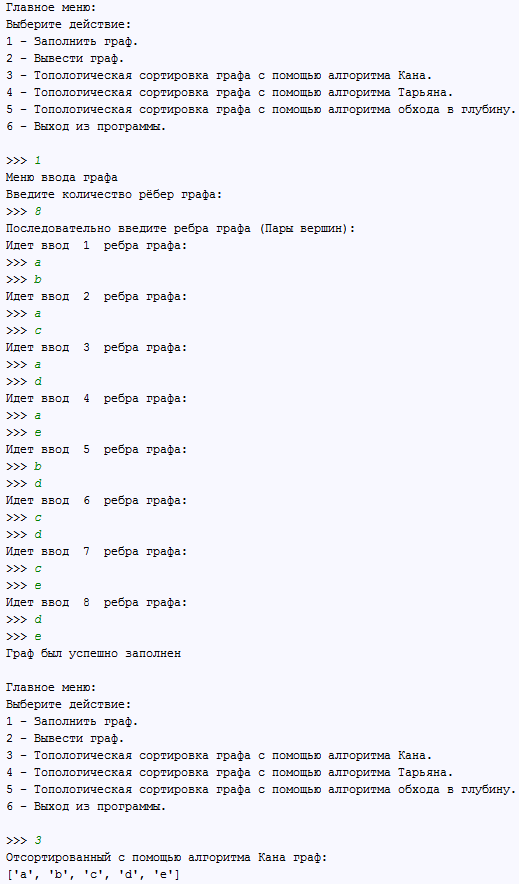


Рис. 8 – Тестирование алгоритма Кана в нормальных условиях.

Теперь следует протестировать представленный алгоритм в условиях, когда исходный граф содержит в себе контур(цикл):

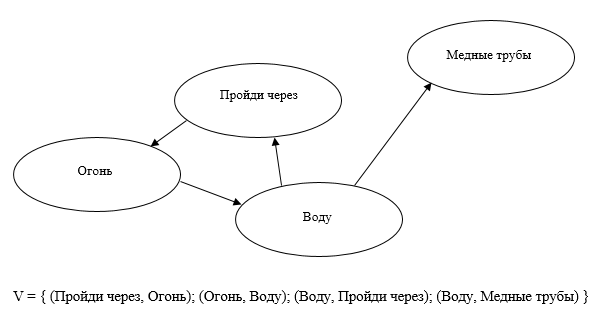


Рис. 9 – Граф, содержащий цикл.

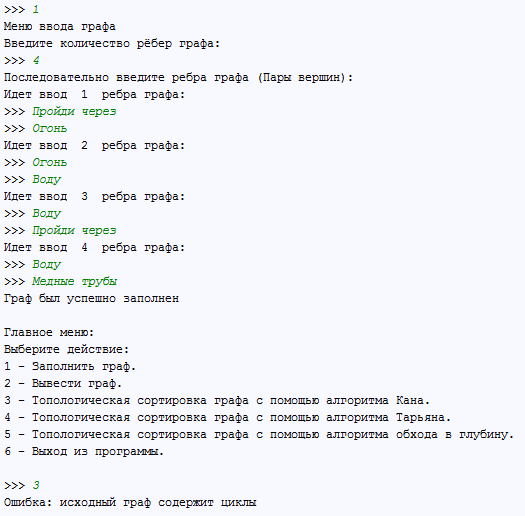


Рис. 9 – Программа сообщила об ошибке, так как граф содержит цикл.

## 4. АЛГОРИТМ ТАРЬЯНА

### 4.1 Описание алгоритма

Данный алгоритм представляет собой усовершенствованную версию алгоритма обхода в глубину. Он был разработан Р. Тарьяном в 1976 году. Этот алгоритм базируется на идее поиска компонент сильной связности в ориентированном граф и работает за линейное время. Ориентированный граф называется сильно связным, если хотя бы две из его вершин являются сильно связными. Две вершины v и u любого графа сильно связны, если существует ориентированный путь из v в u и ориентированный путь из u в v. Орграф, не принадлежащий к классу сильно связных графов, содержит некоторый набор сильно связных компонент, и некоторый набор ориентированных ребер, идущих от одной компоненты к другой. Любая вершина орграфа сильно связна сама с собой.

Данный алгоритм основывается на двух утверждениях:

1. Вершины рассматриваются в обратном топологическом порядке, поэтому в конце рекурсивной функции для исходной вершины не будет встречено ни одной вершины из той же сильной компоненты, так как все вершины, достижимые из исходной, уже обработаны.
2. Обратные связи в дереве дают второй путь из одной вершины в другую и связывают сильные компоненты.

Алгоритм Тарьяна можно понимать как модификацию алгоритма поиска в глубину, в котором при посещении вершины и окончании обработки вершины выполняются дополнительные действия. Посещение вершины происходит при движении от корня к листьям, а окончание обработки вершины — на обратном пути. При посещении вершины она проталкивается во вспомогательный стек, а выталкивается при окончании обработки. Представленная версия алгоритма может топологически сортировать направленные графы, содержащие в себе циклы - это является одним из главных его преимуществ по сравнению с алгоритмом стандартного обхода в глубину, что был рассмотрен выше.

### 4.2 Кодирование алгоритма

1. # Топологическая сортировка с помощью алгоритма Тарьяна
2. **def** Tarjan\_Topological\_Sort(self):
3. # Проверка на введённость графа
4. **if** self.\_edges\_value > 0:
5. # Копия графа
6. Graph = self.\_edges
7. # Счётчик индексов
8. Index\_Counter = List(); Index\_Counter.Append(0)
9. # Стэк хранения преемников текущей вершины
10. Stack = StackList()
11. # Словарь индексов
12. Index = {}
13. # Словарь посещенных вершин
14. Low\_Links = {}
15. # Список отсортированных вершин
16. Сonnected\_Components = List()
18. # Функция поиска преемников пройденной вершины
19. **def** Visited\_Vertex\_Sucsessors\_Search(Vertex):
20. # Шаг 1: Установим индекс глубины для данной вершины
21. # как наименьший неиспользуемый индекс
22. Index[Vertex] = Index\_Counter.ElementAt(0)
23. Low\_Links[Vertex] = Index\_Counter.ElementAt(0)
24. Index\_Counter.ReplaceAt(0, Index\_Counter.ElementAt(0) + 1)
25. Stack.Push(Vertex)
27. # Шаг 2: Рассмотрим преемников вершины
28. **try**:
29. Successors = Graph[Vertex]
30. **except**:
31. Successors = List()
33. **for** Successor **in** Successors:
34. **if** Successor **not** **in** Low\_Links:
35. # Преемник ранее не посещался, поэтому рекурсивно вызываем его обход по преемникам
36. Visited\_Vertex\_Sucsessors\_Search(Successor)
37. Low\_Links[Vertex] = min(Low\_Links[Vertex], Low\_Links[Successor])
38. **elif** Successor **in** Stack:
39. # Преемник находится в стеке, следовательно он сильно связан с текущей вершиной
40. Low\_Links[Vertex] = min(Low\_Links[Vertex], Index[Successor])
42. # Шаг 3: Если текущая вершина - корневая, то заполним стек сильно связанных компонент
43. **if** Low\_Links[Vertex] == Index[Vertex]:
44. # Наполняем список отсортированных вершин
45. **while** True:
46. Successor = Stack.Pop()
47. Сonnected\_Components.Append(Successor)
48. **if** Successor == Vertex:
49. **break**
51. # Цикличный вызов функции поиска сильносвязанных элементов
52. **for** Vertex **in** Graph:
53. **if** Vertex **not** **in** Low\_Links:
54. Visited\_Vertex\_Sucsessors\_Search(Vertex)
56. **print**("Отсортированный с помощью алгоритма Тарьяна граф: ")
57. **print**(Сonnected\_Components)
58. **print**("")
59. **else**:
60. **print**("Ошибка: граф не был введен")
61. **print**("")

### 4.3 Тестирование алгоритма

Для начала, протестируем граф, что был изображен на Рис. 9, чтобы наглядно продемонстрировать преимущества данного алгоритма:

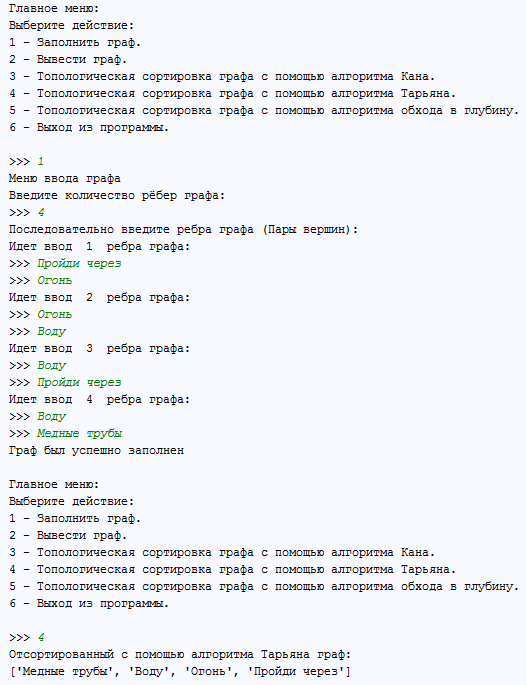


Рис. 10 – Демонстрация работы алгоритма Тарьяна для графа, содержащего цикл. Вершины графа выведены в рекурсивном порядке, так же заметно, что цикл, что содержал в себе граф, был раскрыт и обработан.

Теперь можно протестировать данный алгоритм на более сложном графе, например на том, что будет представлен ниже:

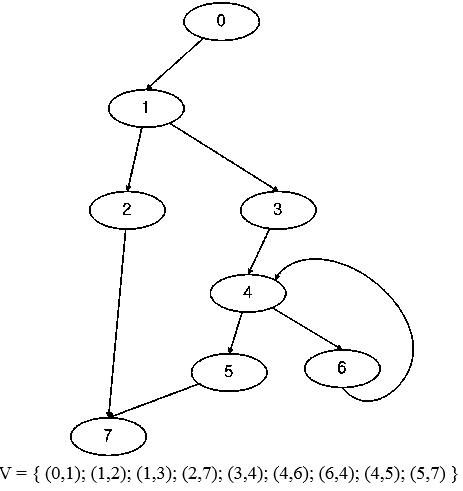


Рис. 11 – Более сложный граф для демонстрации работы сортировки Тарьяна.

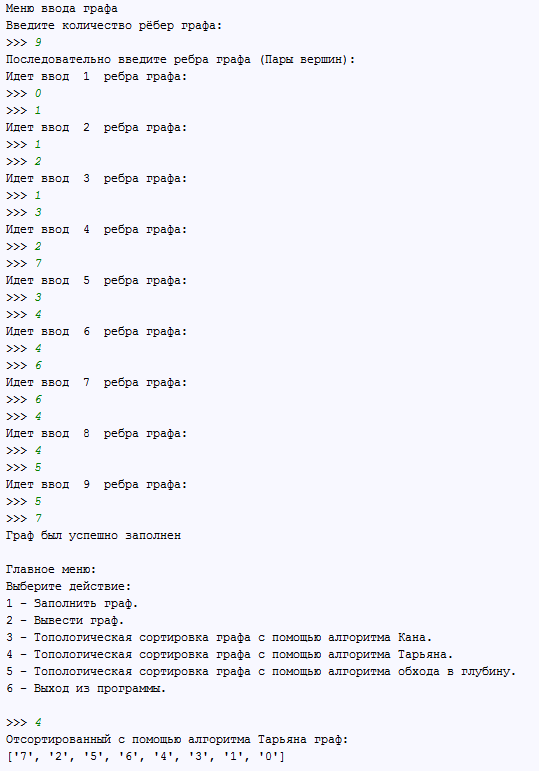


Рис. 12 – Ещё один отсортированный данным алгоритмом граф, по рекурсивному порядку выведенных вершин можно проследить ход алгоритмом по вершинам исходного графа.

## 5 ОПИСАНИЕ ИНТЕРФЕЙСА ПРОГРАММЫ

### 5.1 Ввод

Ввод графов в программе осуществляется путем выбора соответствующей функции главного меню программы. После этого программа попросит пользователя ввести число ребер графа, которое может быть исключительно положительным. После того, как будет указано количество ребер, программа попросит их циклично ввести. Ребром графа может являться любое слово на русском или английском языках, любое число, а так же комбинация цифр и символов. В случае некорректного ввода программа сообщит пользователю об ошибке и попросит повторить ввод.

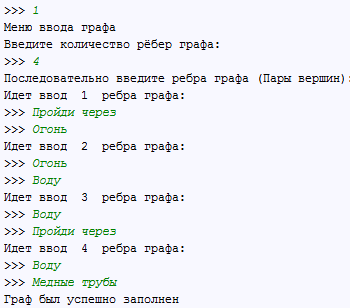


Рис. 13 – Пример ввода графа.

### 5.2 Вывод

Вывод введенного графа осуществляется путем последовательного вывода в столбец множества вершин исходного графа, причем если граф окажется пуст, программа сообщит о возникновении ошибок.

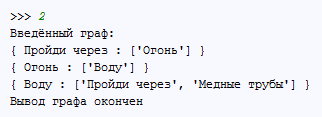


Рис. 14 – Пример вывода графа.

### 5.3 Организация работы сортировок

Выбор алгоритма сортировки графа происходит при помощи меню ввода, после чего при условии заполненности графа выводится её результат.

# ЗАКЛЮЧЕНИЕ

Цель данной курсовой работы – разработка программного приложения для демонстрации работы нескольких алгоритмов топологической сортировки для ориентированных графов.

Были выполнены следующие задачи:

* Изучены стандарты оформления кода на языке Python
* Изучены алгоритмы топологической сортировки и их программная реализация
* Спроектирована структура программного приложения
* Реализовано программное приложение средствами языка Python

# СПИСОК ИСПОЛЬЗОВАННОЙ ЛИТЕРАТУРЫ

1. Роберт Л. Круз, «Структуры данных и проектирование программ», Бином. Лаборатория знаний, издание 2015 года
2. Марк Лутц, «Изучаем Python. Подробный справочник», Символ-Плюс, издание 2015 года.
3. Седжвик Роберт «Алгоритмы на C++. Анализ структуры данных. Сортировка. Поиск. Алгоритмы на графах. Руководство», Диасофт, издание 2016 года.
4. Никлаус Вирт, «Алгоритмы + структуры данных = программы», ДМК Пресс, издание 2016 года.
5. Александр Омельченко, «Теория графов», МЦНМО, издание 2018 года.
6. Майкл Доусон, «Программируем на Python», Питер, издание 2018 года.
7. Стивен Скиена «Алгоритмы. Руководство по разработке», БХВ-Петербург, издание 2017 года.
8. «Инструкция по организации и проведению курсового проектирования», Московский Технологический университет, издание 2018 года.
9. Описание простейшего алгоритма обхода в глубину: <http://neerc.ifmo.ru/wiki/index.php?title=Использование_обхода_в_глубину_для_топологической_сортировки>

# ПРИЛОЖЕНИЯ

## 1. ИСХОДНЫЙ КОД ПРОГРАММЫ

### Node.py

1. # Универсальный класс-узел:
2. **class** Node(object):
3. # Конструтор:
4. **def** \_\_init\_\_(self, data  = None, \_next = None, \_prev = None):
5. self.\_next = \_next
6. self.data = data
7. self.\_prev = \_prev
9. # Приведение к строке:
10. **def** \_\_repr\_\_(self):
11. **return** repr(self.data)

### LQS.py

1. # Импорт класса-узла
2. **from** Node **import** Node
4. # Класс-шаблон для создания списков, очередей и стеков
5. **class** LQS(object):
6. # Конструктор:
7. **def** \_\_init\_\_(self, length = 0):
8. self.\_head = None
9. self.\_tail = None
10. self.length = 0
12. # Преобразование к строке
13. **def** \_\_repr\_\_(self):
14. Nodes = []
15. Current = self.\_head
16. **if** **not** self.\_head:
17. **return** '[' + ', '.join(Nodes) + ']'
18. **if** **not** Current.\_next:
19. Nodes.append(repr(Current))
20. **else**:
21. Current = self.\_head
22. **while** Current:
23. Nodes.append(repr(Current))
24. Current = Current.\_next
25. **return** '[' + ', '.join(Nodes) + ']'
27. # Включение свойства iterable
28. **def** \_\_iter\_\_(self):
29. Сurrent = self.\_head
30. **while** Сurrent:
31. **yield** Сurrent.data
32. Сurrent = Сurrent.\_next
34. # Удаление объекта:
35. **def** Clear(self):
36. self.\_head = None

### List.py

1. # Импорт класс узла
2. **from** Node **import** Node
3. # Импорт класса-шаблона
4. **from** LQS **import** LQS
6. # Класс двусвязного списка
7. **class** List(LQS):
8. # Вставить элемент в конец списка
9. **def** Append(self, data):
10. **if** **not** self.\_head:
11. self.\_head = self.\_tail = Node(data = data)
12. self.length += 1
13. **return**
14. Current = self.\_head
15. **while** Current.\_next:
16. Current = Current.\_next
17. Current.\_next = self.\_tail = Node(data = data, \_prev=Current)
18. self.length += 1
20. # Вставить элемент в начало списка
21. **def** Prepend(self, data):
22. new\_head = Node(data = data, \_next = self.\_head)
23. **if** **not** self.\_head:
24. self.\_head = self.\_tail = Node(data = data)
25. self.length += 1
26. **elif** self.\_head:
27. self.\_head.\_prev = new\_head
28. self.\_head = new\_head
29. Current = self.\_head
30. **while** Current:
31. Current = Current.\_next
32. self.\_tail = Current
33. self.length += 1
35. # Найти элемент в списке
36. **def** \_Find(self, key):
37. Current = self.\_head
38. **while** Current **and** Current.data != key:
39. Current = Current.\_next
40. **return** Current
42. # Удалить узел из списка
43. **def** \_Delete(self, Node):
44. **if** Node.\_prev:
45. Node.\_prev.\_next = Node.\_next
46. **if** Node.\_next:
47. Node.\_next.\_prev = Node.\_prev
48. **if** Node **is** self.\_head:
49. self.\_head = Node.\_next
50. **if** Node **is** self.\_tail:
51. self.\_tail = Node.\_prev
52. Node.\_prev = None
53. Node.\_next = None
55. # Удалить первое вхождение элемента из списка
56. **def** Remove(self, key):
57. **if** self.length == 0:
58. **return**
59. **else**:
60. Element = self.\_Find(key)
61. **if** **not** Element:
62. **return**
63. self.\_Delete(Element)
64. self.length -= 1
66. # Удалить первый элемент списка
67. **def** DeleteFirst(self):
68. **if** self.\_head != None:
69. self.\_Delete(self.\_head)
70. self.length -= 1
71. **else**:
72. **return**
74. # Удалить последний элемент списка
75. **def** DeleteLast(self):
76. **if** self.length == 0:
77. **return**
78. **elif** self.length == 1:
79. self.\_Delete(self.\_head)
80. self.length -= 1
81. **elif** self.\_tail:
82. self.\_Delete(self.\_tail)
83. self.length -= 1
85. # Перевернуть список
86. **def** Reverse(self):
87. **if** self.length <= 1:
88. **return**
89. **else**:
90. self.\_tail = self.\_head
91. Current = self.\_head
92. Previous\_Node = None
93. **while** Current:
94. Previous\_Node = Current.\_prev
95. Current.\_prev = Current.\_next
96. Current.\_next = Previous\_Node
97. Current = Current.\_prev
98. self.\_head = Previous\_Node.\_prev
100. # Вернуть элемент по индексу:
101. **def** ElementAt(self, index):
102. **if** self.length == 0 **or** index > self.length - 1:
103. **return**
104. Count = 0
105. Current = self.\_head
106. **while** Count < index:
107. Current = Current.\_next
108. Count += 1
109. **return** Current.data
111. # Найти индекс первого вхождения элемента
112. **def** IndexOf(self, key):
113. Current = self.\_head
114. Count = 0
115. **while** Current **and** Current.data != key:
116. Current = Current.\_next
117. Count += 1
118. **return** Count
120. # Удалить элемент с указанной позиции
121. **def** RemoveAt(self, index):
122. **if** self.length == 0 **or** index > self.length - 1:
123. **return**
124. Count = 1
125. index += 1
126. Current = self.\_head
127. **while** Current:
128. Previous\_Node = Current.\_prev
129. Next\_Node = Current.\_next
130. **if** Count == index:
131. **if** Previous\_Node:
132. Previous\_Node.\_next = Next\_Node
133. **if** Next\_Node:
134. Next\_Node.prev = Previous\_Node
135. **else**:
136. self.\_head = Next\_Node
137. **if** Next\_Node:
138. Next\_Node.prev = None
139. **return** self
140. Current = Next\_Node
141. Count += 1
143. # Заменить элемент по индексу
144. **def** ReplaceAt(self, index, data):
145. **if** self.length == 0 **or** index > self.length - 1:
146. **return**
147. Count = 0
148. Current = self.\_head
149. **while** Count < index:
150. Current = Current.\_next
151. Count += 1
152. Current.data = data

### StackList.py

1. # Импорт класс узла:
2. **from** Node **import** Node
3. # Импорт класса-шаблона:
4. **from** LQS **import** LQS
6. # Класс стека:
7. **class** StackList(LQS):
8. # Поместить в стек:
9. **def** Push(self, data):
10. **if** **not** self.\_head:
11. self.\_head = self.\_tail = Node(data = data)
12. self.length += 1
13. **return**
14. **else**:
15. Current = self.\_head
16. **while** Current.\_next:
17. Current = Current.\_next
18. Current.\_next = self.\_tail = Node(data = data, \_prev = Current)
19. self.length += 1
21. # Удалить из стека:
22. **def** Pop(self):
23. **if** self.length == 0:
24. **return**
25. **else**:
26. Current = self.\_head
27. **if** **not** Current.\_prev **and** **not** Current.\_next:
28. Head\_Buffer = self.\_head
29. self.\_head = None
30. self.length -= 1
31. **return** Head\_Buffer.data
32. **else**:
33. **while** Current.\_next:
34. Current = Current.\_next
35. Current\_Buffer = Current.\_prev.\_next
36. Current.\_prev.\_next = None
37. self.length -= 1
38. **return** Current\_Buffer.data
40. # Получить элемент из стека:
41. **def** Peek(self):
42. **if** self.length == 0:
43. **return**
44. **else**:
45. Current = self.\_head
46. **if** **not** Current.\_prev **and** **not** Current.\_next:
47. **return** self.\_head.data
48. **else**:
49. **while** Current.\_next:
50. Current = Current.\_next
51. **return** Current.\_prev.\_next.data

### QueueList.py

1. # Класс очереди:
2. **class** QueueList(LQS):
3. # Поместить в очередь:
4. **def** Enqueue(self, data):
5. **if** **not** self.\_head:
6. self.\_head = self.\_tail = Node(data = data)
7. self.length += 1
8. **return**
9. **else**:
10. Current = self.\_head
11. **while** Current.\_next:
12. Current = Current.\_next
13. Current.\_next = self.\_tail = Node(data = data, \_prev = Current)
14. self.length += 1
16. # Удалить из очереди:
17. **def** Dequeue(self):
18. **if** self.length == 0:
19. **return**
20. **else**:
21. **if** **not** self.\_head.\_next:
22. Head\_Buffer = self.\_head
23. self.\_head = self.\_head.\_next
24. self.length -= 1
25. **return** Head\_Buffer.data
26. **else**:
27. Current = Head\_Buffer = self.\_head
28. **while** Current:
29. Current = Current.\_next
30. self.\_head = Head\_Buffer.\_next
31. Current = self.\_head
32. self.length -= 1
33. **return** Head\_Buffer.data
35. # Получить элемент из очереди:
36. **def** Peek(self):
37. **if** self.\_head:
38. **return** self.\_head.data

### Graph.py

1. # Импорт регулярных выражений
2. **import** re
3. # Импорт списка
4. **from** List **import** List
5. # Импорт очереди
6. **from** QueueList **import** QueueList
7. # Импорт стека
8. **from** StackList **import** StackList
10. # Класс графа
11. **class** Graph(object):
12. # Конструктор
13. **def** \_\_init\_\_(self, edges\_value = 0 , vertex\_value = 0):
14. # Граф строится на основе словаря
15. self.\_edges = {}
16. # Количество рёбер
17. self.\_edges\_value = edges\_value
19. # Метод заполнения графа
20. **def** Input\_Graph(self):
21. self.\_edges = {}
22. **print**("Меню ввода графа")
23. **print**("Введите количество рёбер графа: ")
24. self.\_edges\_value = self.\_Graph\_Edges\_Input\_Controller()
25. **print**("Последовательно введите ребра графа (Пары вершин): ")
26. **for** i **in** range(self.\_edges\_value):
27. **print**("Идет ввод ", i + 1, " ребра графа: ")
28. Current\_Node = self.\_Lexema\_Filter()
29. Next\_Node = self.\_Lexema\_Filter()
30. Current\_List = List()
31. Current\_List.Append(Next\_Node)
32. **if** Current\_Node **in** self.\_edges:
33. self.\_edges[Current\_Node].Append(Next\_Node)
34. **else**:
35. self.\_edges[Current\_Node] = Current\_List
36. **print**("Граф был успешно заполнен")
37. **print**("")
39. # Вывод графа
40. **def** Output\_Graph(self):
41. **if** self.\_edges\_value > 0:
42. **print**("Введённый граф: ")
43. **for** key **in** self.\_edges:
44. **print**("{",key, ":",self.\_edges[key],"}")
45. **print**("Вывод графа окончен ")
46. **print**("")
47. **else**:
48. **print**("Граф не был введён ")
49. **print**("")
51. # Защита от некорректного ввода чисел
52. **def** \_Graph\_Edges\_Input\_Controller(self):
53. **while** True:
54. **try**:
55. Value = int(input(">>> "))
56. **if** Value > 0:
57. **return** Value
58. **else**:
59. **print**("Ошибка ввода")
60. **print**("")
61. **except** ValueError:
62. **print**("Ошибка ввода")
63. **print**("")
65. # Фильтр лексем
66. **def** \_Lexema\_Filter(self):
67. # Паттерн всего, что  является лексемой
68. Pattern = r'[\.a-zA-Zа-яА-я0-9]+'
69. **while** True:
70. Lexem = input(">>> ")
71. **if** re.search(Pattern, Lexem):
72. **return** Lexem
73. **else**:
74. **print**("Ошибка ввода")
75. **print**("")
77. # Топологическая сортировка с помощью алгоритма Кана
78. **def** Kahn\_Topological\_Sort(self):
79. # Проверка на введённость графа
80. **if** self.\_edges\_value > 0:
81. # Копия графа
82. Graph = self.\_edges
83. # Список с отсортированным графом
84. Topological\_Sorted\_List = List()
85. # Очередь вершин с нулем входящих связей
86. Zero\_In\_Degree\_Vertex\_Queue = QueueList()
87. # Словарь входящих связей соседних вершин
88. In\_Degree = {Key: 0 **for** Key **in** Graph}
90. # Шаг 1: Обойти граф и посчитать связи соседних вершин
91. **for** Key **in** Graph:
92. **for** Value **in** Graph[Key]:
93. # Если вершина не была обозначена ранее, добавим её
94. **if** Value **in** In\_Degree:
95. In\_Degree[Value] += 1
96. **else**:
97. In\_Degree[Value] = 1
99. # Шаг 2: Обновим изначальный граф
100. **for** Key **in** In\_Degree:
101. **if** Key **in** In\_Degree **and** Key **not** **in** Graph:
102. Graph[Key] = List()
104. # Шаг 3: Поиск вершин с нулем входящих связей
105. **for** Key **in** In\_Degree:
106. **if** In\_Degree[Key] == 0:
107. Zero\_In\_Degree\_Vertex\_Queue.Enqueue(Key)
109. # Шаг 4: Обработка узлов с нулем вхоядищих связей
110. **while** Zero\_In\_Degree\_Vertex\_Queue.length > 0:
111. Value = Zero\_In\_Degree\_Vertex\_Queue.Dequeue()
112. Topological\_Sorted\_List.Append(Value)
114. # Шаг 5: Обновим словарь входящих связей
115. **for** Neighbour **in** Graph[Value]:
116. In\_Degree[Neighbour] -= 1
117. **if** (In\_Degree[Neighbour] == 0):
118. Zero\_In\_Degree\_Vertex\_Queue.Enqueue(Neighbour)
120. # Шаг 6: Проверка на циклы и вывод отсортированного графа
121. **if** Topological\_Sorted\_List.length == len(Graph):
122. **print**("Отсортированный с помощью алгоритма Кана граф: ")
123. **print**(Topological\_Sorted\_List)
124. **print**("")
125. **else**:
126. **print**("Ошибка: исходный граф содержит циклы")
127. **print**("")
128. **else**:
129. **print**("Ошибка: граф не был введен")
130. **print**("")
132. # Топологическая сортировка с помощью алгоритма Тарьяна
133. **def** Tarjan\_Topological\_Sort(self):
134. # Проверка на введённость графа
135. **if** self.\_edges\_value > 0:
136. # Копия графа
137. Graph = self.\_edges
138. # Счётчик индексов
139. Index\_Counter = List(); Index\_Counter.Append(0)
140. # Стэк хранения преемников текущей вершины
141. Stack = StackList()
142. # Словарь индексов
143. Index = {}
144. # Словарь посещенных вершин
145. Low\_Links = {}
146. # Список отсортированных вершин
147. Сonnected\_Components = List()
149. # Функция поиска преемников пройденной вершины
150. **def** Visited\_Vertex\_Sucsessors\_Search(Vertex):
151. # Шаг 1: Установим индекс глубины для данной вершины
152. # как наименьший неиспользуемый индекс
153. Index[Vertex] = Index\_Counter.ElementAt(0)
154. Low\_Links[Vertex] = Index\_Counter.ElementAt(0)
155. Index\_Counter.ReplaceAt(0, Index\_Counter.ElementAt(0) + 1)
156. Stack.Push(Vertex)
158. # Шаг 2: Рассмотрим преемников вершины
159. **try**:
160. Successors = Graph[Vertex]
161. **except**:
162. Successors = List()
164. **for** Successor **in** Successors:
165. **if** Successor **not** **in** Low\_Links:
166. # Преемник ранее не посещался, поэтому рекурсивно вызываем его обход по преемникам
167. Visited\_Vertex\_Sucsessors\_Search(Successor)
168. Low\_Links[Vertex] = min(Low\_Links[Vertex], Low\_Links[Successor])
169. **elif** Successor **in** Stack:
170. # Преемник находится в стеке, следовательно он сильно связан с текущей вершиной
171. Low\_Links[Vertex] = min(Low\_Links[Vertex], Index[Successor])
173. # Шаг 3: Если текущая вершина - корневая, то заполним стек сильно связанных компонент
174. **if** Low\_Links[Vertex] == Index[Vertex]:
175. # Наполняем список отсортированных вершин
176. **while** True:
177. Successor = Stack.Pop()
178. Сonnected\_Components.Append(Successor)
179. **if** Successor == Vertex:
180. **break**
182. # Цикличный вызов функции поиска сильносвязанных элементов
183. **for** Vertex **in** Graph:
184. **if** Vertex **not** **in** Low\_Links:
185. Visited\_Vertex\_Sucsessors\_Search(Vertex)
187. **print**("Отсортированный с помощью алгоритма Тарьяна граф: ")
188. **print**(Сonnected\_Components)
189. **print**("")
191. **else**:
192. **print**("Ошибка: граф не был введен")
193. **print**("")
195. # Топологическая сортировка обходом в глубину
196. **def** Deep\_First\_Topological\_Sort(self):
197. # Проверка на введённость графа
198. **if** self.\_edges\_value > 0:
199. # Граф
200. Graph = self.\_edges
201. # Дополним конечные вершины графа
202. BufferList = List()
203. **for** Vertex **in** Graph:
204. **for** Element **in** Graph[Vertex]:
205. **if** Element **not** **in** Graph.keys() **and** Element **not** **in** BufferList:
206. BufferList.Append(Element)
207. **for** Element **in** BufferList:
208. Graph[Element] = List()
209. # Результирующий список
210. Result = List()
211. # Словарь "цветов" вершин
212. Colors = { Vertex: "White" **for** Vertex **in** Graph}
213. # Переменная состояния найденного цикла
214. Founded\_Cycle = False
216. # Цикл обхода вершин графа
217. **for** Vertex **in** Graph:
218. # Если вершина не посещена, она будет белого цвета
219. **if** Colors[Vertex] == "White":
220. self.\_Deep\_First\_Sort\_Recursion(Graph, Vertex, Colors, Result, Founded\_Cycle)
221. # Если найден цикл - прекращаем
222. **if** Founded\_Cycle:
223. **break**
224. **if** Founded\_Cycle **or** Result.length != Graph.\_\_len\_\_():
225. **print**("Сортировка невозможна, так как исходный граф содержит цикл")
226. **print**("")
227. **return**
228. Result.Reverse()
229. **print**("Топологически отсортированный граф: ")
230. **print**(Result)
231. **print**("")
232. **else**:
233. **print**("Ошибка: граф не был введен")
234. **print**("")
236. # Рекурсивный цикл обхода вершин для DFS
237. **def** \_Deep\_First\_Sort\_Recursion(self, Graph, Vertex, Colors, Result, Founded\_ Cycle):
238. # Найден цикл - возврат
239. **if** Founded\_Cycle:
240. **return**
241. # Обозначим посещенную вершину серым цветом
242. Colors[Vertex] = "Gray"
243. # Цикл обхода преемников посещенной вершины
244. **for** SubVertex **in** Graph[Vertex]:
245. # Если из посещенной вершины попали в посещенную - обнаружен цикл
246. **if** Colors[SubVertex] == "Gray":
247. Founded\_Cycle = True
248. **return**
249. # Если вершина-преемник не посещена, то рекурсивно обойдем и ее
250. **if** Colors[SubVertex] == "White":
251. self.\_Deep\_First\_Sort\_Recursion(Graph, SubVertex, Colors, Result, Founded\_Cycle)
252. # Когда дальнейший обход невозможен, покрасим вершину в черный цвет
253. # и добавим в результирующий список
254. Colors[Vertex] = "Black"
255. Result.Append(Vertex)

### Interface.py

1. # Класс интерфейса
2. **class** Interface:
3. # Конструктор
4. **def** \_\_init\_\_(self):
5. **pass**
7. #Главное меню:
8. **def** Interface\_Main(self):
9. **print** ("Главное меню:")
10. **print** ("Выберите действие:")
11. **print** ("1 - Заполнить граф.")
12. **print** ("2 - Вывести граф.")
13. **print** ("3 - Топологическая сортировка графа с помощью алгоритма Кана.")
14. **print** ("4 - Топологическая сортировка графа с помощью алгоритма Тарьяна.")
15. **print** ("5 - Топологическая сортировка графа с помощью алгоритма обхода в глубину.")
16. **print** ("6 - Выход из программы.")
17. **pass**
19. #Защита от некорректного ввода:
20. **def** Input\_Controller(self):
21. **while** True:
22. **try**:
23. **print**("")
24. Value = int(input(">>> "))
25. **if** Value > 0:
26. **return** Value
27. **else**:
28. **print**("Ошибка ввода!")
29. **except** ValueError:
30. **print**("Ошибка ввода!")

### Main.py

1. **print** ("Курсовая работа по СиАОД: Топологическая сортировка графов. Лисовой А.А. ИКБО-12-17")
2. # Импорт графа
3. **from** Graph **import** Graph
4. # Импорт интерфейса
5. **from** Interface **import** Interface
7. # Консольное меню программы
8. Menu = Interface()
9. # Вызов метода отображения главного меню
10. Menu.Interface\_Main()
11. # Создание объекта класса Граф
12. Graph = Graph()
14. # Цикл обработки команд
15. **while** True:
16. # Ввод выбранного действия
17. Step = Menu.Input\_Controller()
19. # Ввод графа с клавиатуры
20. **if** Step == 1:
21. Graph.Input\_Graph()
22. Menu.Interface\_Main()
24. # Вывод графа
25. **elif** Step == 2:
26. Graph.Output\_Graph()
27. Menu.Interface\_Main()
29. # Сортировка алгоритмом Кана
30. **elif** Step == 3:
31. Graph.Kahn\_Topological\_Sort()
32. Menu.Interface\_Main()
34. # Сортировка алгоритмом Тарьяна
35. **elif** Step == 4:
36. Graph.Tarjan\_Topological\_Sort()
37. Menu.Interface\_Main()
39. # Сортировка алгоритмом обхода в глубину
40. **elif** Step == 5:
41. Graph.Deep\_First\_Topological\_Sort()
42. Menu.Interface\_Main()
44. # Выход из программы
45. **elif** Step == 6:
46. **print**("Программа завершена")
47. **break**
49. # Остальные случаи
50. **else**:
51. **print** ("Ошибка ввода")
52. **print**("")

## 2. ПРЕЗЕНТАЦИЯ



Рис. 15 – Слайд 1 презентации к курсовой работе.



Рис. 16 – Слайд 2 презентации к курсовой работе.

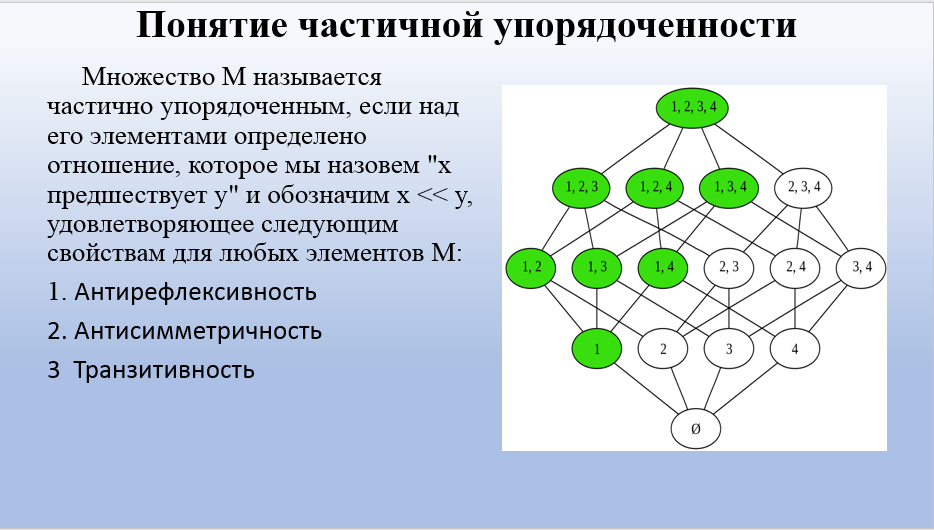


Рис. 17 – Слайд 3 презентации к курсовой работе.



Рис. 18 – Слайд 4 презентации к курсовой работе.



Рис. 19 – Слайд 5 презентации к курсовой работе.

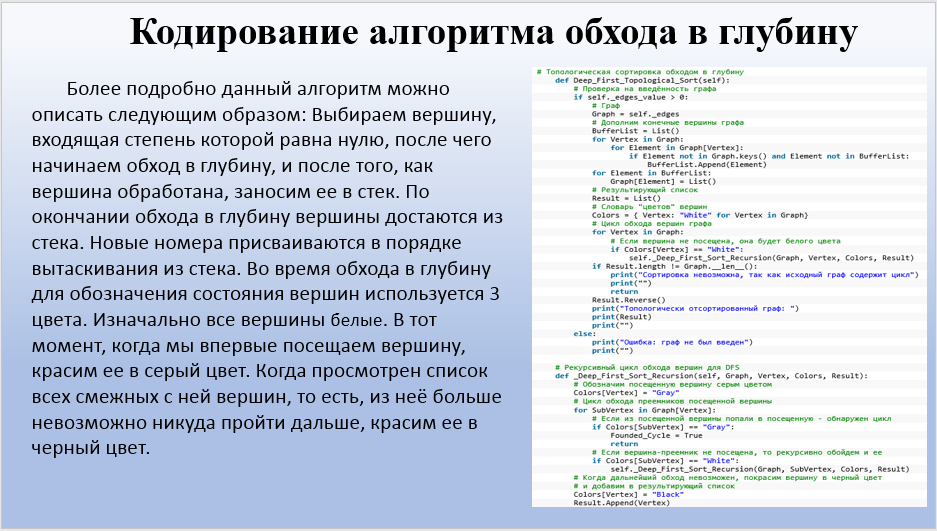


Рис. 20 – Слайд 6 презентации к курсовой работе.



Рис. 21 – Слайд 7 презентации к курсовой работе.

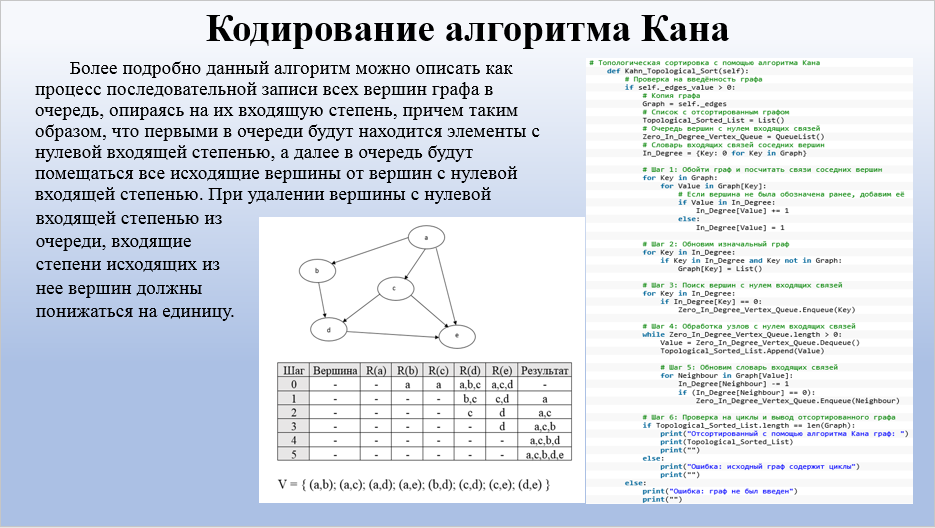


Рис. 22 – Слайд 8 презентации к курсовой работе.



Рис. 23 – Слайд 9 презентации к курсовой работе.

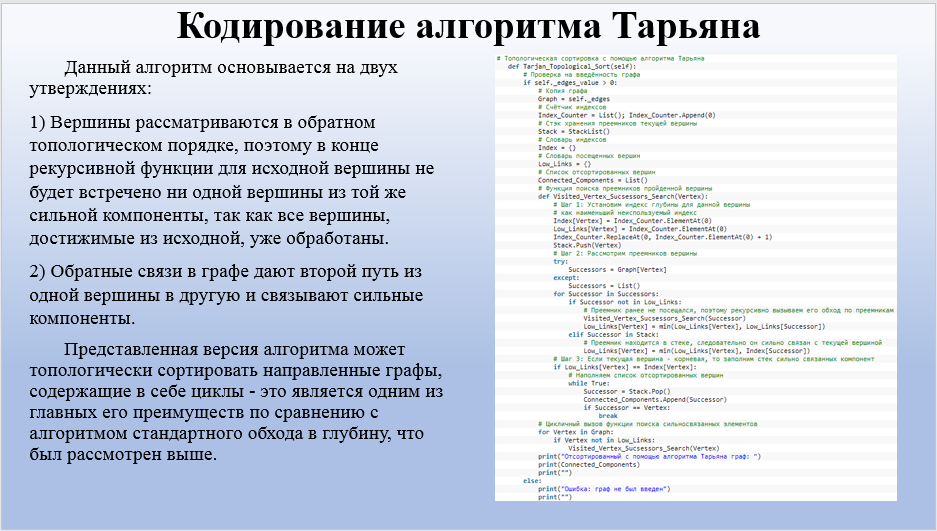


Рис. 24 – Слайд 10 презентации к курсовой работе.

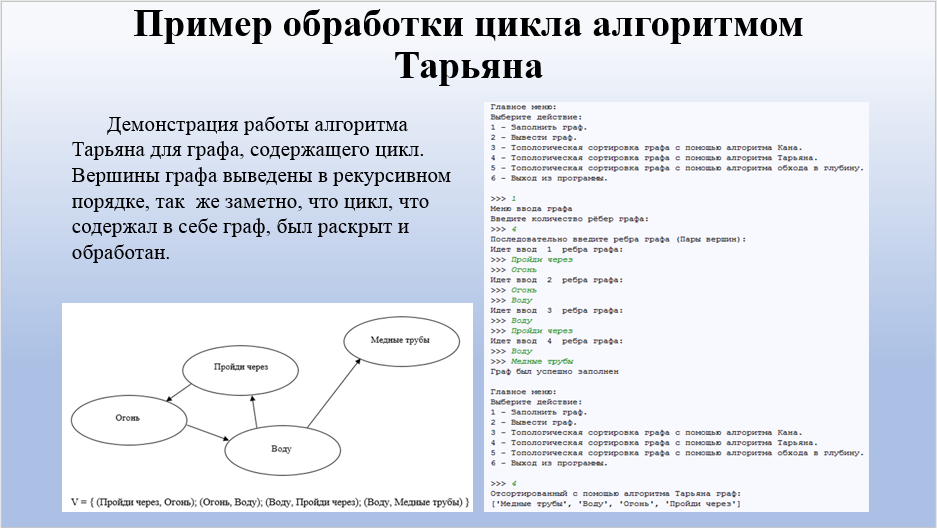


Рис. 25 – Слайд 11 презентации к курсовой работе.

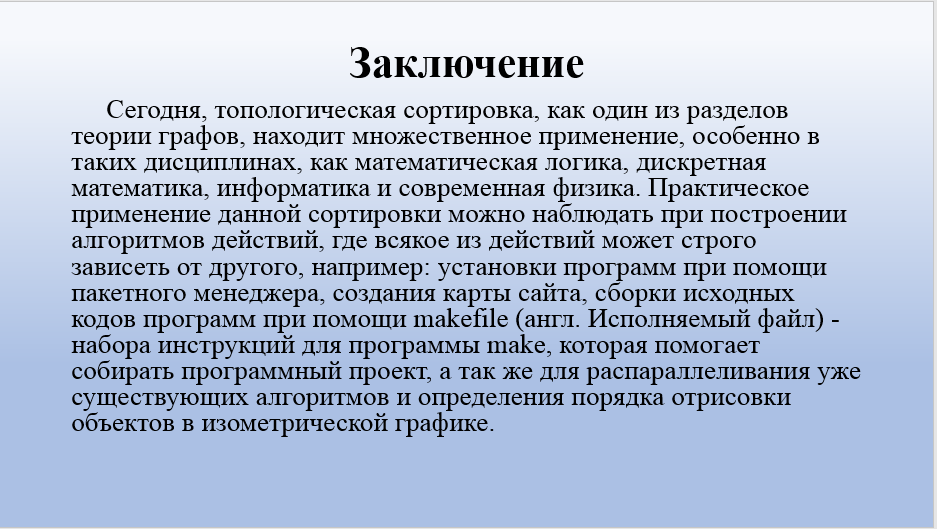


Рис. 26 – Слайд 12 презентации к курсовой работе.