«Министерство образования и науки Российской Федерации Федеральное государственное бюджетное образовательное учреждение высшего образования «Алтайский государственный технический университет им. И.И. Ползунова»

Факультет информационных технологий

Кафедра прикладной математики

Отчёт защищён с оценкой\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Преподаватель Астахова А.В.

«\_\_\_\_\_»\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_2022 г.

Отчёт

Лабораторной работе №2

«Управление проектами. Расчет параметров событий и работ сетевого  
графика. Определение критического пути»

Студент группы ПИ 92 В.М. Шульпов

Преподаватель доцент, к. эк. н. Астахова А.В.

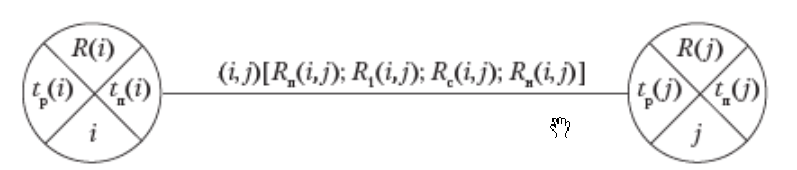
Барнаул 2022

**Задание**.  
1. Изучить параграф 2.5 (лекция 2) и Приложение к работе.  
(см. в Илиас «Лекция 2» и файл с материалами учебника: часть 2)  
2. Разработать тестовые примеры для выполнения работы.  
3. Разработать алгоритм расчета параметров событий и работ СГ с учетом  
требований Приложения к работе и данного пункта задания.  
При разработке алгоритма и программы не должны использоваться  
стандартные библиотеки работы с графами и программы расчета сетевых  
графиков.  
Составить и отладить программу на выбранном языке программирования,  
которая обращается к программе, разработанной в лабораторной работе 2, и  
выводит на экран монитора:  
– исходный список работ СГ;  
– таблицу «Параметры событий СГ»:

|  |  |  |
| --- | --- | --- |
| Событие | Сроки свершения события | Резерв времени события |
| ранний | поздний |  |

–- таблицу «Параметры работ СГ»:

|  |  |  |
| --- | --- | --- |
| Работа | Продолжительность работы | Резервы времени работы |
| полный | независимый |  |

– список работ всех критических путей СГ, которые определяются с учетом  
необходимого и достаточного условия отнесения пути к критическому;  
– длину критического пути.  
4. Оформить отчет, внеся все необходимые комментарии в программу. Защитить  
работу   
Пусть *i* – индекс, соответствующий предшествующему событию,  
*j* – индекс, соответствующий последующему событию,  
*(i, j)* – работа, связанная с событиями *i* и *j*, начинающаяся событием *i* и  
завершающаяся событием *j*,  
*τ(i, j)* – продолжительность выполнения работы *(i, j)* в единицах измерения  
времени*.***1. Параметры событий СГ  
Алгоритм прямого хода** определения параметров «Ранний срок свершения  
событий» ***t  
р (i)*** – при обходе вершин графа от начальной вершины к конечной с учетом  
слоев графа (то есть путем обхода графа в ширину):  
для начального события *t р (i = нач.) := 0;*для событий *j* : *t р (j) = max [ t р (i) + τ(i, j)]  
i***Алгоритм обратного хода** определения параметров «Поздний срок свершения  
событий» ***t п (j)*** и резерв времени свершения события *R(i)* – при обходе вершин графа от  
конечной вершины к начальной:  
для конечного события *t п (j = кон.)* := *t р (j = кон.);*для событий *i* : *t п (i ) = min [ t п (j) – τ(i, j)];  
j*для начального события должно получиться: *t п (i = нач.)* = 0. Если это не так, то  
где-то ошибка в алгоритме.  
*R(i) = t п (i )– t р (i )***Длина критического пути**: T = *t п (j = кон.)* := *t р (j = кон.)***2. Параметры работ СГ. Основные формулы**Полный резерв времени работы:  
*Rп (i,j)* = *t п (j) – t р (i)– τ(i, j)***Необходимое и достаточное условие** принадлежности работ критическому пути  
СГ: если для всех работ *(i,j)* СГ, принадлежащих некоторому полному пути, ***Rп (i,j)=0****,* то  
данный путь является критическим.  
Очевидно, таких путей в одном СГ может быть несколько.  
Независимый резерв времени работы :  
*Rн (i,j)* = *t р (j) – t п (i)– τ(i, j)*Если *Rн (i,j)* > 0, то с данной работой связан **подвижной состав ресурсов**, которые  
при оптимизации СГ можно: либо изъять, либо закрепить за работами критического пути.

Структура графа:

self.fictive\_start\_top = -100 *# шифр фиктивной стартовой вершины*self.fictive\_end\_top = -101 *# шифр фиктивной конечной вершины*self.graph\_table = {**'arc\_start'**: [],  
 **'arc\_end'**: [],  
 **'weight'**: [],  
 **'is\_visited'**: [],  
 *# 'layer': [],* } *# таблица, содержащая дуги и веса графа*self.struct\_graph\_table = {**'arc\_start'**: [],  
 **'arc\_end'**: [],  
 **'weight'**: [],  
 **'is\_visited'**: [],  
 **'full\_time\_reserve'**: [],  
 **'independent\_time\_reserve'**: [],  
 } *# упоряденная таблица, содержащая дуги и веса графа  
# таблица, содержащая шифр события, его ранний срок, поздний срок и резервное время  
# количество входящих и выходящих из события работ*self.graph\_events = {**'event'**: [],  
 **'early\_term'**: [],  
 **'late\_term'**: [],  
 **'reserve\_time'**: [],  
 *# 'in\_works\_num': [],  
 # 'out\_works\_num': [],* **'in\_works'**: [],  
 **'out\_works'**: [],  
 **'layer'**: [],  
 **'visited'**: []  
 }  
self.first\_top = **None** *# первая вершина графа*self.last\_top = **None** *# последняя вершина графа*self.works\_num = 0 *# количество работ графа*self.struct\_graph\_works\_num = 0 *# количество работ упорядоченного графа*self.current\_way = [] *# текущий путь для вывода всех полных путей*

0) инициализация и получение исходного графа из файла происходит в конструкторе \_\_init\_\_(self, file\_in)

1) метод print\_row(self, row\_index, sorted\_graph=False)печатает строку таблицу по индексу

2) метод print\_graph(self, sorted\_graph=False)печатает всю таблицу графа, используя функцию print\_row. Есть параметр функции, который указывает какой граф выводить (упорядоченный или нет).

3) метод add\_row\_in\_graph\_table(self, start, end, weight, is\_visited=False, adding\_is\_in\_sort=False) добавляет строку в таблицу графа. Есть параметр функции, который указывает в какой граф добавлять (упорядоченный или нет). При добавлении метод увеличивает счётчик количества строк соответствующего графа на 1.

4) метод def copy\_row\_to\_struct\_graph\_table(self, index) копирует строку по индексу из неупорядоченного графа и добавляет её в конец упорядоченного. Метод вызывает метод add\_row\_in\_graph\_table.

5) метод delete\_row\_from\_graph\_table(self, index) удаляет строку из неупорядоченного графа. Уменьшает счётчик количества строк в неупорядоченном графе на 1.

6) метод search\_first\_top(self) ищет начальную вершину СГ.

Проходит во внешнем цикле по всем строкам таблицы путей графа  
Во внутреннем цикле проверяется, не является ли стартовой (в неё входит дуга, кроме петель) текущая вершина. Если найдена стартовая вершина (и не посещенная), проверяется, первая ли она такая. Если первая, то происходит запоминание её шифра и переход к следующей вершине. Если стартовая вершина уже существует, то происходит добавление фиктивной вершины (добавляется дополнительная запись в таблицу СГ.

7) метод delete\_top\_loops(self, index) проверяет строку на наличие петли и удаляет строку, если петля есть. Возвращает счётчик цикла: если было удаление, то возвращает 0, чтобы запустить цикл сначала, если не было удаления, то возвращает тот же индекс цикла и ничего не изменяется. (так как нет goto)

8) метод check\_duplication(self, i, j) проверяет наличие дублирования строки (с одинаковым и разным весом). При разных весах предлагает выбрать, какую работу удалить. При одинаковом удаляет автоматически вторую (разницы какую удалять нет). Метод аналогично возвращает счётчик цикла для запуска цикла сначала. (так как нет goto).

9) метод search\_last\_top(self) работает аналогично методу с поиском первой вершины. В нём происходит поиск последней вершины. Если вершина не одна, создается фиктивная конечная вершина.

10) метод optimize\_graph(self) вызывает методы delete\_top\_loops, check\_duplication, search\_last\_top и управляет порядком их вызова. Когда нужн повторно запустить метод, когда нет он понимает по возвращаемым значениям из этих функций (так как нет goto).

11) метод struct\_graph(self) упорядочивает граф, используя очередь (проходит граф в ширину). Новый упорядоченный граф хранится другом объекте.

12) метод search\_BFS(self) проходит по графу в ширину

Если последний элемент в текущем пути равен последнему элементу в графе, запускается цикл печати пути, удаляется последняя вершина из текущего пути и происходит выход из функции.

Иначе происходит вход в цикл( i от 0 до количества строк в таблице графа). Если стартовая вершина i-ой работы равна последней работе в текущем пути, то добавляем в текущий путь конечную вершину i-ой работы и рекурсивно запускаем search\_BFS

После этого цикла удаляется последняя вершина из текущего пути.

13) метод search\_full\_ways(self) находит все полные пути графа, вызывая рекурсивный метод recursive\_search(self). Перед этим вызовом находится первая строка, в которой хранится первая вершина. Добавляет эту вершину в текущий путь.

15) **def** append\_new\_event\_in\_events\_graph(self, event):  
 *""" добавлние новой вершины в graph\_events, инит параметров нулями """*

16) **def** swap\_events(self, i, j):  
 *""" поменять местами строки событий"""*

17) **def** sort\_events\_by\_layers(self):  
 *""" сортировка событий по слоям по возрастанию """*

18) **def** find\_early\_term\_for\_event(self, event):  
 *""" нахождение раннего срока события """*

19) **def** find\_early\_term\_for\_all\_event(self):  
 *""" нахождение раннего срока всех событий """*

20) **def** find\_late\_term\_for\_event(self, event):  
 *""" нахождение позднего срока события """*

21) **def** find\_late\_term\_for\_all\_event(self):  
 *""" нахождение позднего срока всех событий """*

22) **def** find\_time\_reserves\_for\_all\_event(self):  
 *""" нахождение резервов времени для всех событий """*

24) **def** get\_critical\_path\_length(self):  
 *""" получение критического пути из таблицы событий """*

25) **def** find\_time\_reserve\_for\_work(self, index, full=**True**):  
 *""" нахождение полного или независмого временного резерва для работы """  
 # ПОЛНЫЙ ВРЕМЕННОЙ РЕЗЕРВ = раннее время конечного события работы - позднее время начального события работы -  
 # - время выполнения работы (вес работы)  
 # НЕЗАВИСИМЫЙ ВРЕМЕННОЙ РЕЗЕРВ = раннее время начального события работы -  
 # - позднее время конечного события работы - время выполнения работы (вес работы)  
 # full = True, то находится полный временной резерв*

26) **def** find\_time\_reserves\_for\_all\_works(self):  
 *""" нахождение полных и независимых временных резервов для всех работ """*

27) **def** find\_index\_by\_event(self, event):  
 *""" нахождение номера (индекса) события в структуре по его шифру """*

28) **def** check\_vertex\_includes\_only\_prev\_layers\_vertices(self, index):  
 *""" проверить, что в вершину входят только вершины предыдущих слоев """*

29) **def** get\_max\_layer\_form\_prev\_vertices(self, index):  
 *""" получаем максимальный слой предыдущих вершин данной вершины """*

30) **def** find\_vertex\_layers(self):  
 *""" находим слой каждой вершины обходом в ширину"""*

31) **def** find\_ins\_and\_outs(self):  
 *""" находим входящие и исходящие работы у вершин """*

32) **def** find\_all\_vertices(self):  
 *""" нахождение всех вершин графа """*

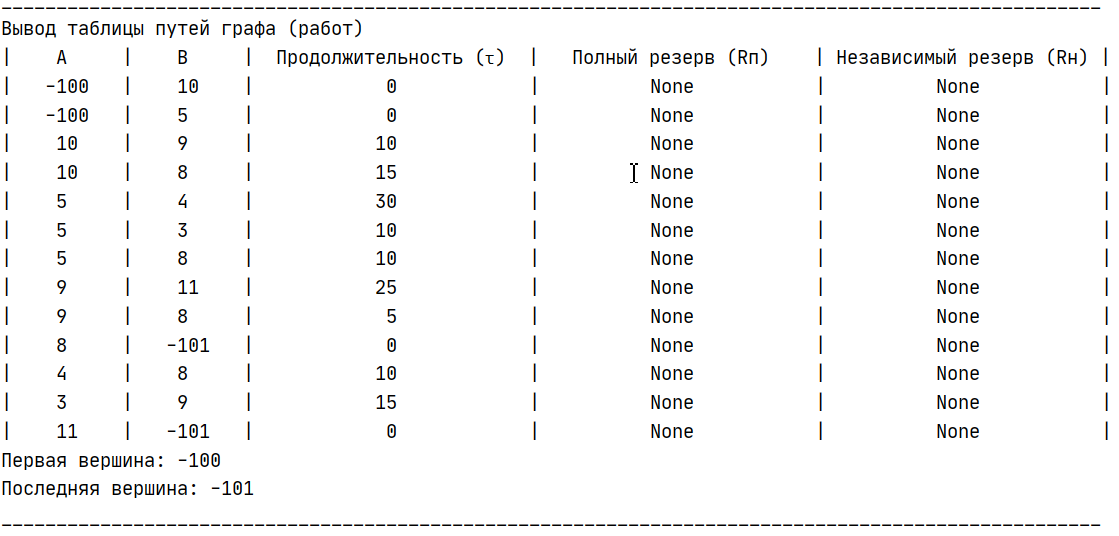
33) **def** recursive\_find\_all\_vertices(self):  
 *""" рекурсивный обход графа """*

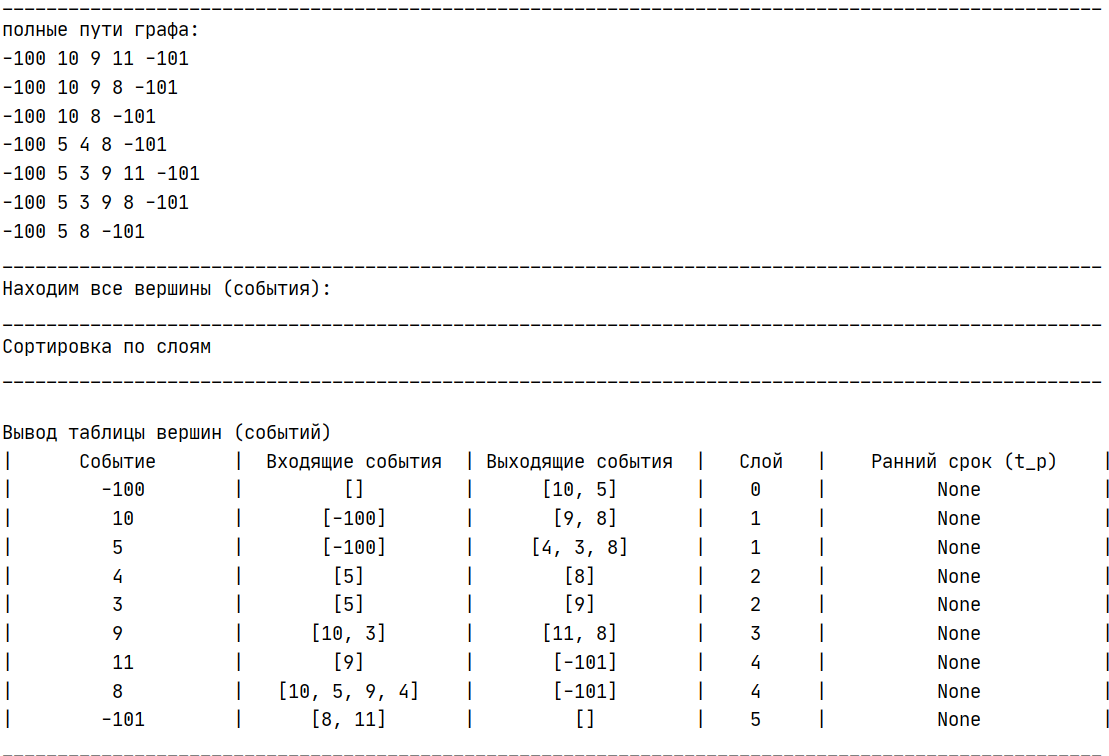
34) код основой программы

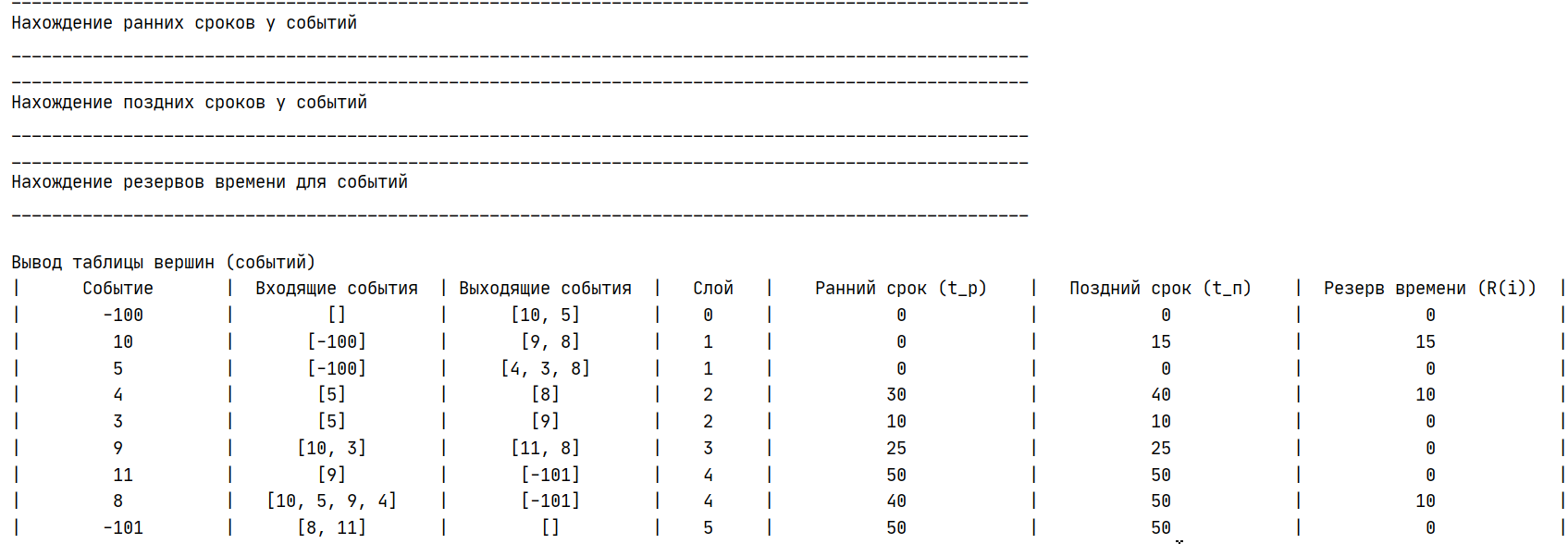
**if** \_\_name\_\_ == **"\_\_main\_\_"**:  
 in\_file = **"input.csv"** graph = Graph(in\_file)  
 graph.print\_graph()  
 *# находим первую вершину* graph.search\_first\_top()  
 *# оптимизируем (удаляем петли, дубли), там же находим конечную вершину* graph.optimize\_graph()  
 graph.print\_graph()  
 *# структурируем граф* graph.struct\_graph()  
 graph.print\_graph(sorted\_graph=**True**)  
 *# находим полные пути* graph.search\_full\_ways()  
 *# находим все вершины* graph.find\_all\_vertices()  
 *# находим входящие и исходящие вершины* graph.find\_ins\_and\_outs()  
 *# находим слои у вершин* graph.find\_vertex\_layers()  
 graph.print\_events\_table()  
 *# сортируем таблицу событий по слоям* graph.sort\_events\_by\_layers()  
 graph.print\_events\_table()  
 *# находим ранние сроки для событий* graph.find\_early\_term\_for\_all\_event()  
 *# находим поздние сроки для событий* graph.find\_late\_term\_for\_all\_event()  
 *# находим временной резерв для событий* graph.find\_time\_reserves\_for\_all\_event()  
 graph.print\_events\_table()  
 *# находим полные и независимые временные резервы для работы* graph.find\_time\_reserves\_for\_all\_works()  
 graph.print\_graph(sorted\_graph=**True**)  
 *# выводим критические пути* graph.search\_full\_ways(check\_critical=**True**)  
 *# находим длину критического пути* critical\_path\_length = graph.get\_critical\_path\_length()

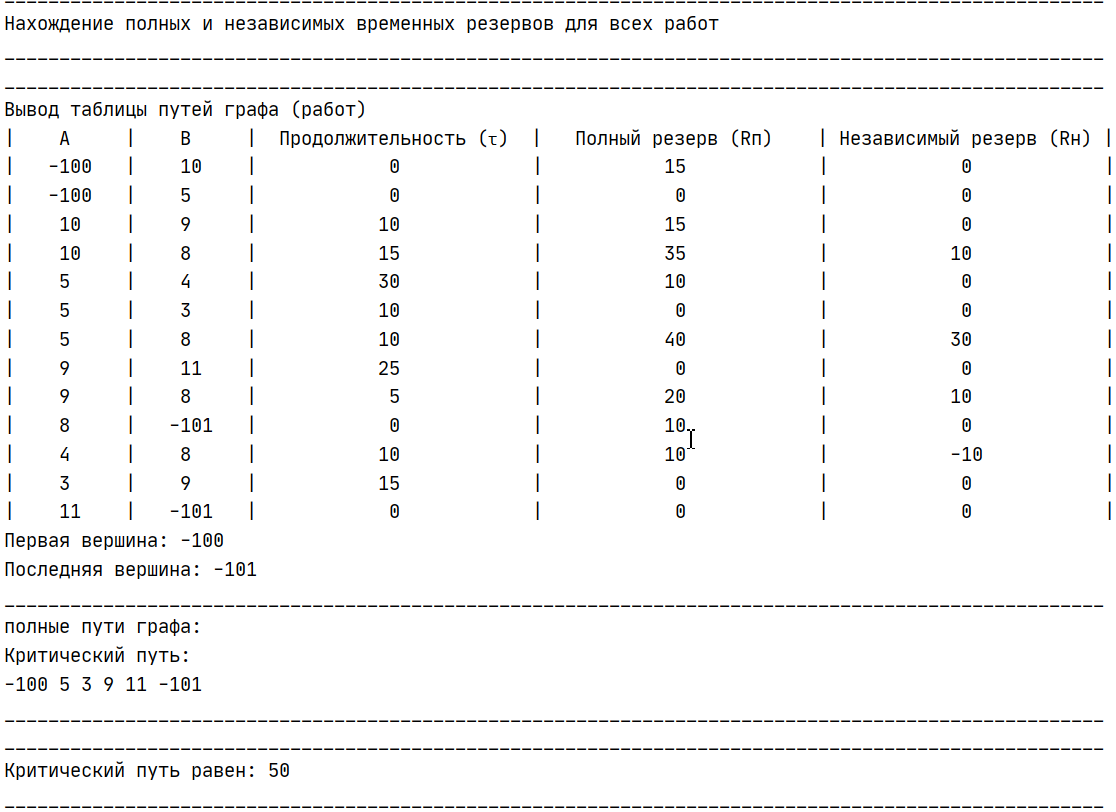
**Тесты:**

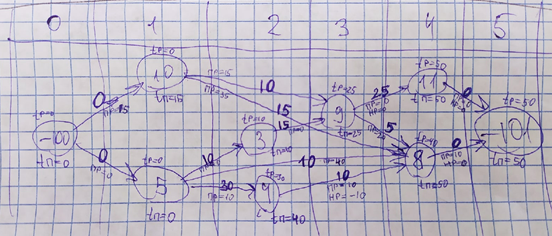
**Тест 1**



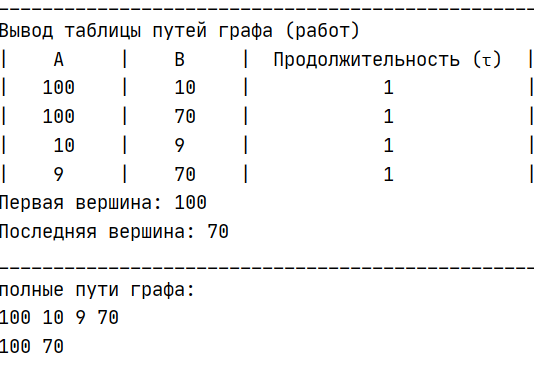


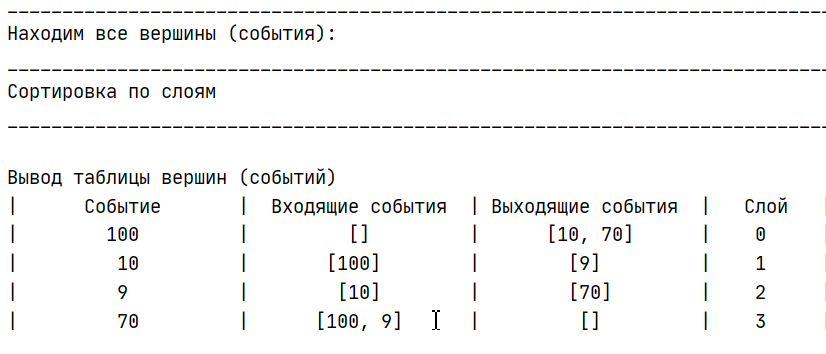


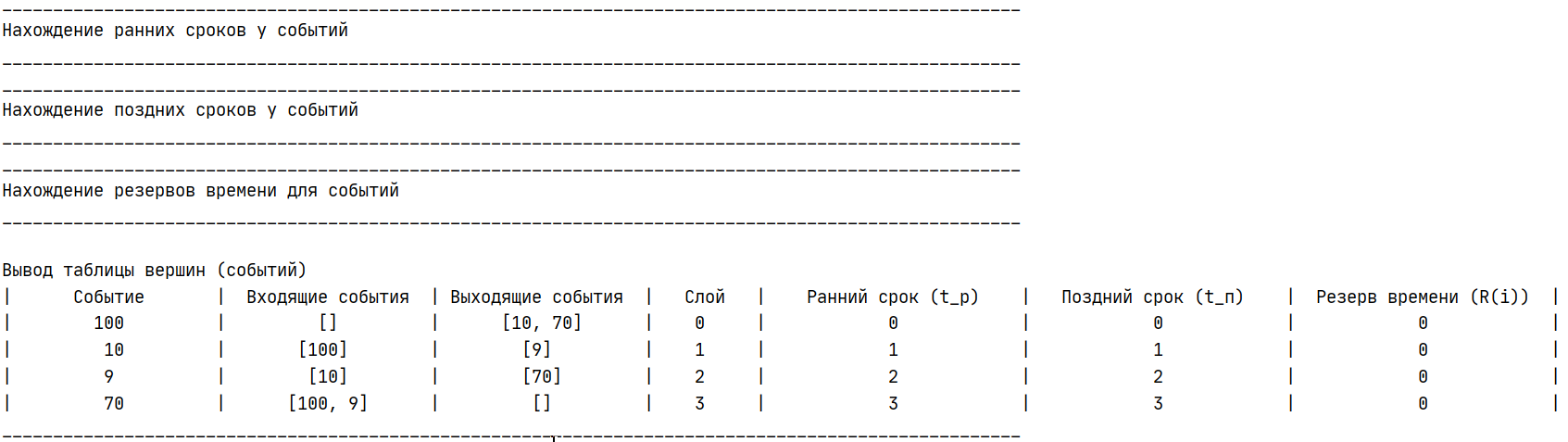


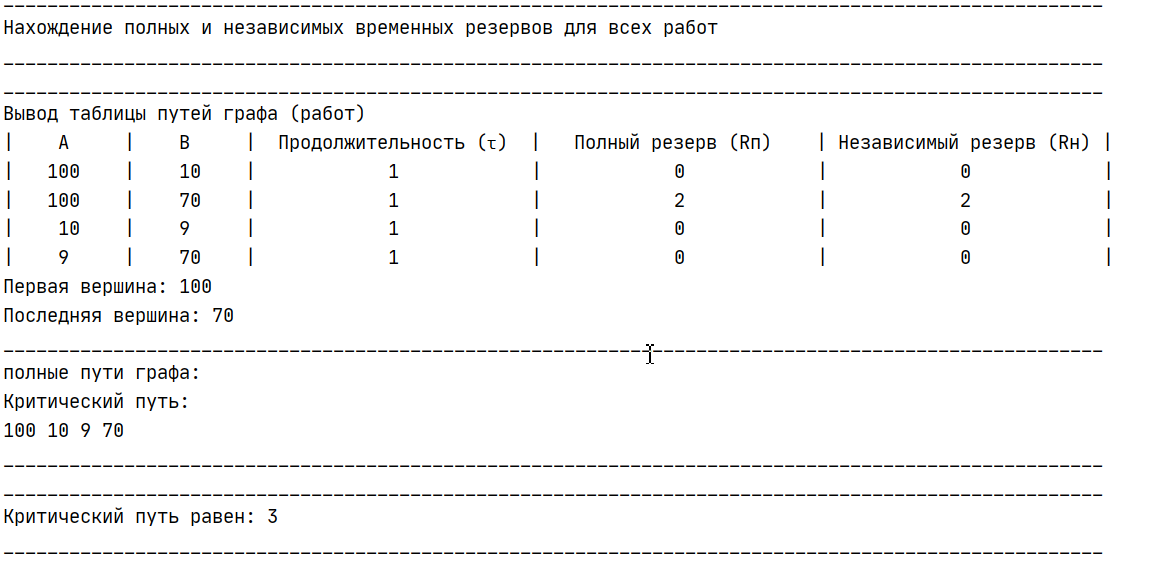
****

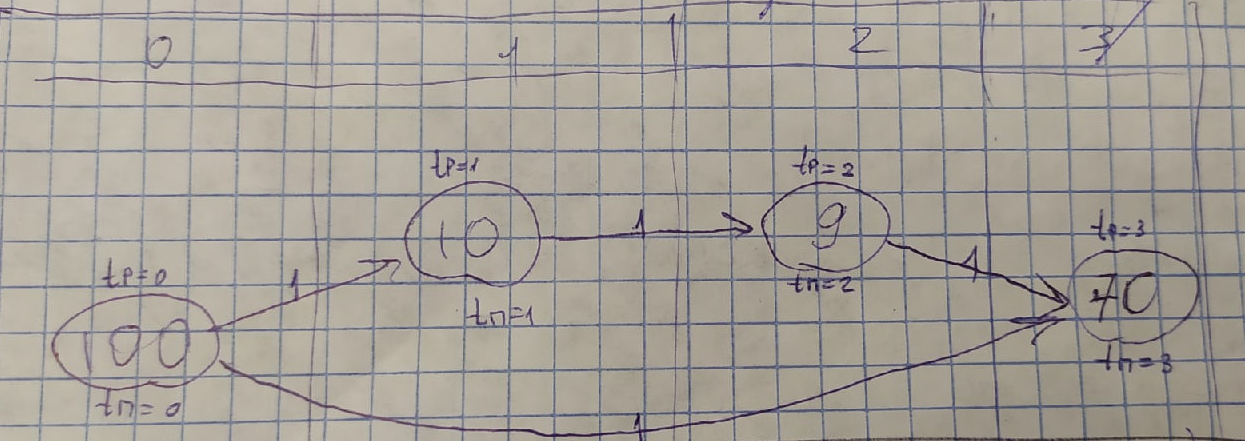
**Тест 2**



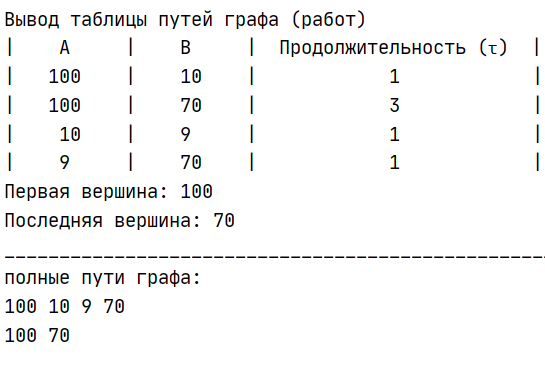


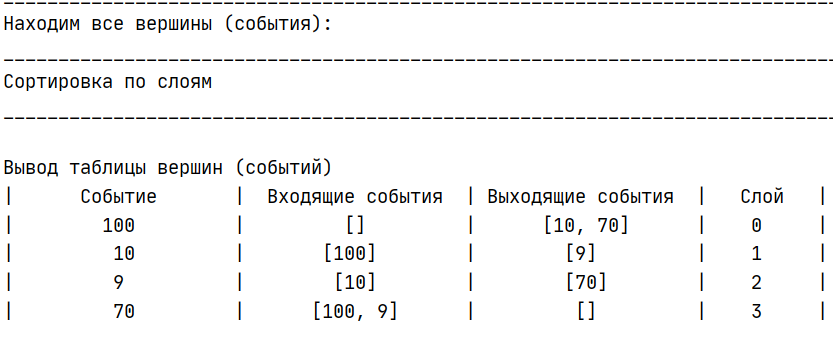


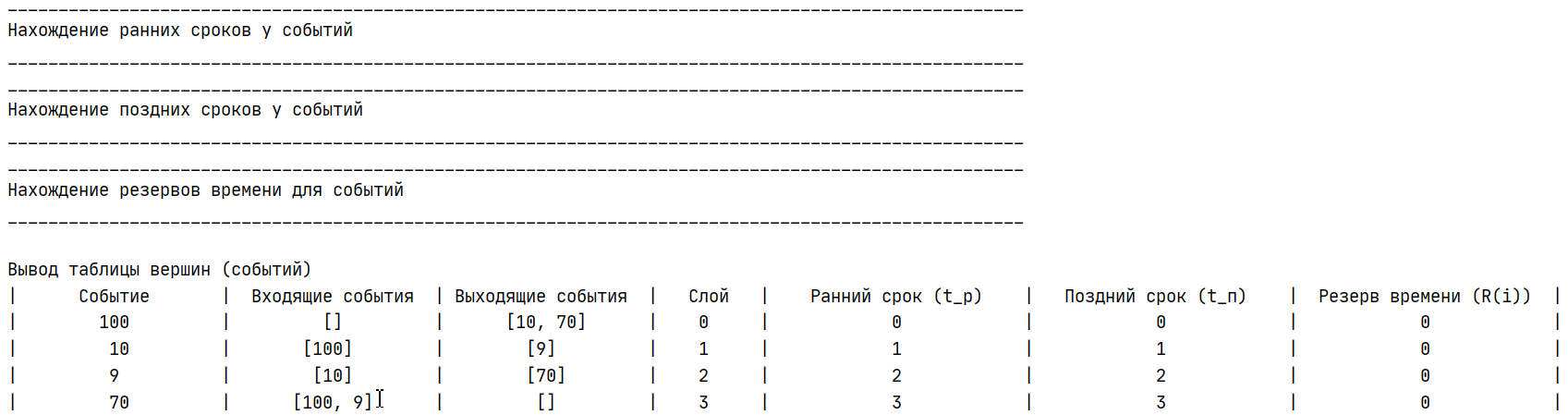


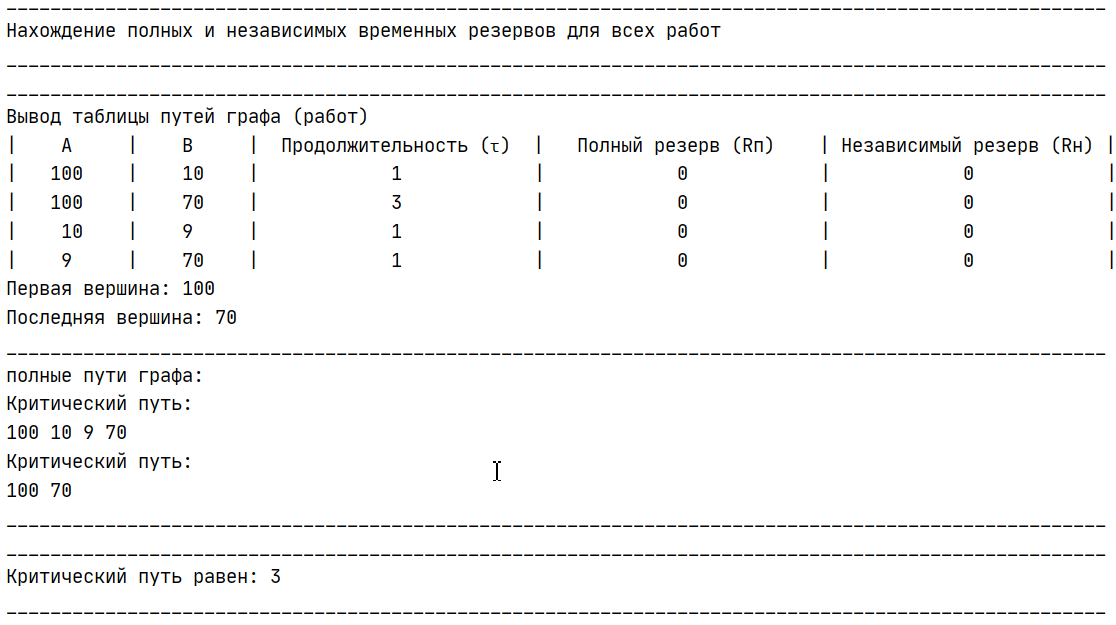
****

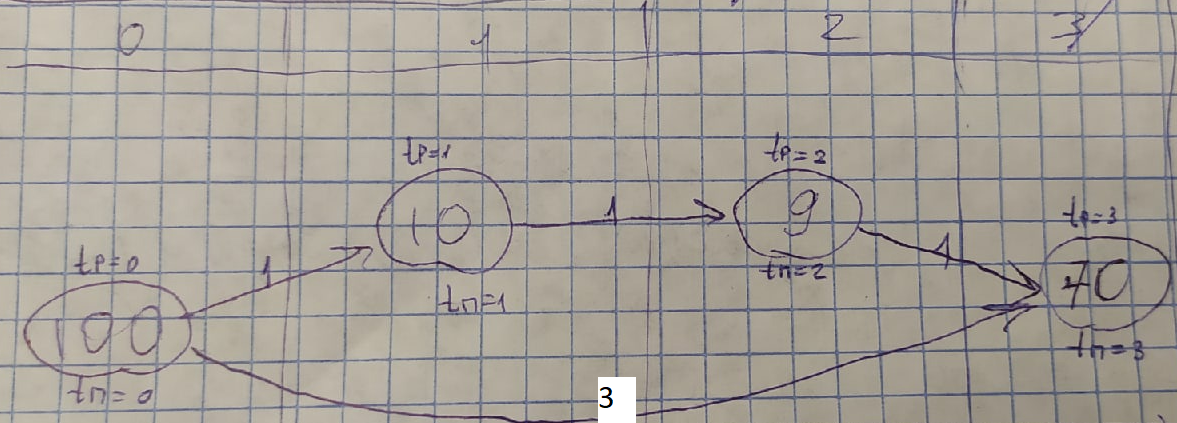
**Тест 3 (2 критических пути)**



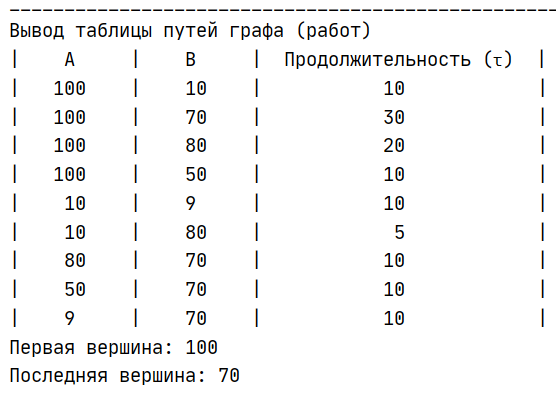


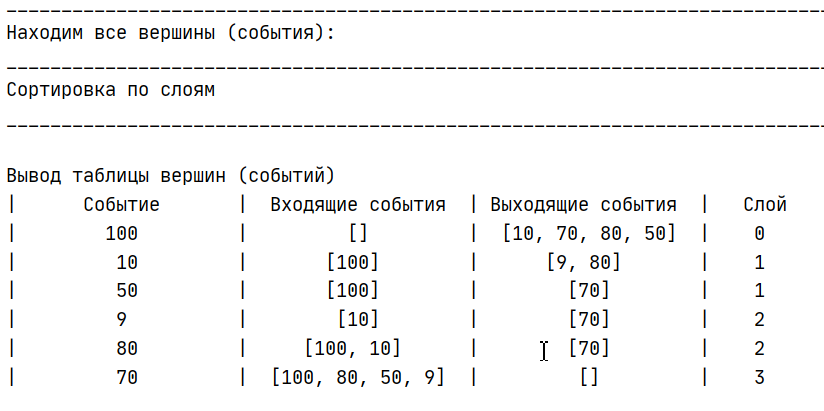


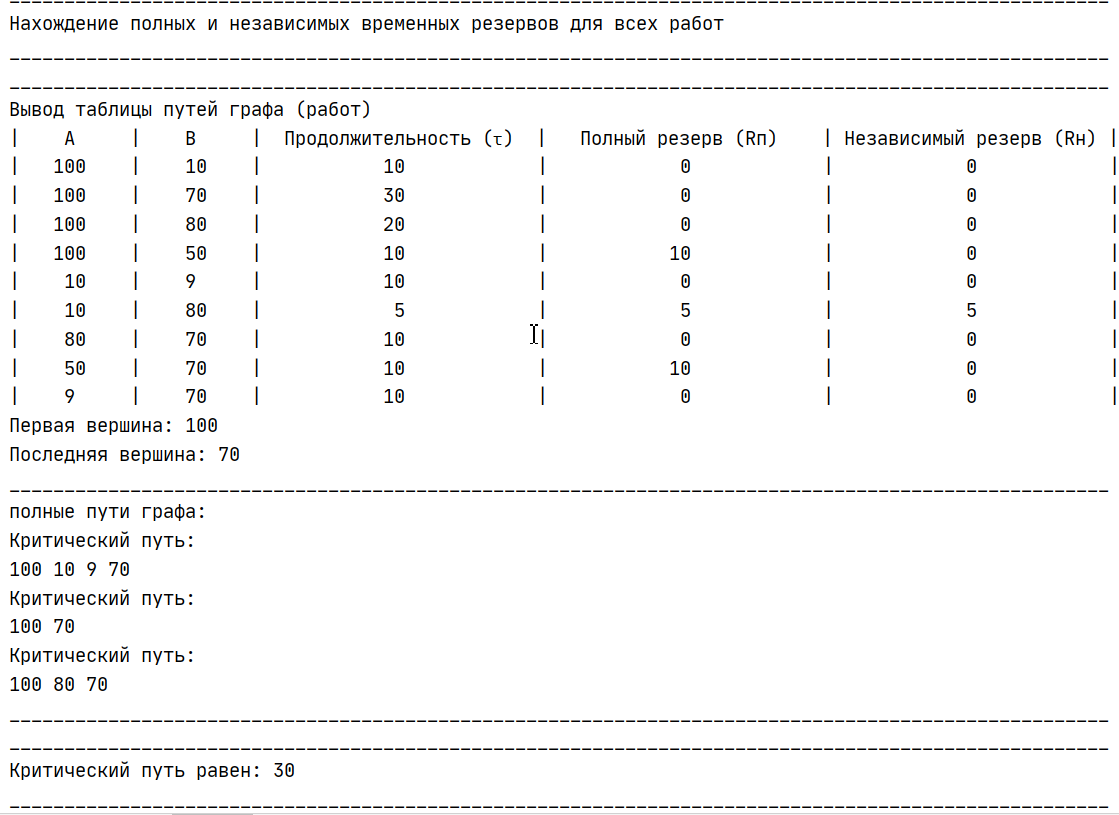


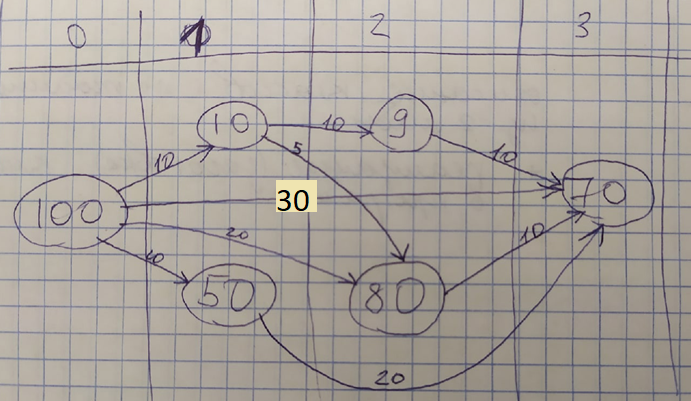
****

**Тест 4 (3 критических пути)**





****

**Код:**

**import** csv  
  
  
**class** Graph:  
  
 **def** \_\_init\_\_(self, file\_in):  
 self.fictive\_start\_top = -100 *# шифр фиктивной стартовой вершины* self.fictive\_end\_top = -101 *# шифр фиктивной конечной вершины* self.graph\_table = {**'arc\_start'**: [],  
 **'arc\_end'**: [],  
 **'weight'**: [],  
 **'is\_visited'**: [],  
 *# 'layer': [],* } *# таблица, содержащая дуги и веса графа* self.struct\_graph\_table = {**'arc\_start'**: [],  
 **'arc\_end'**: [],  
 **'weight'**: [],  
 **'is\_visited'**: [],  
 **'full\_time\_reserve'**: [],  
 **'independent\_time\_reserve'**: [],  
 } *# упоряденная таблица, содержащая дуги и веса графа  
 # таблица, содержащая шифр события, его ранний срок, поздний срок и резервное время  
 # количество входящих и выходящих из события работ* self.graph\_events = {**'event'**: [],  
 **'early\_term'**: [],  
 **'late\_term'**: [],  
 **'reserve\_time'**: [],  
 *# 'in\_works\_num': [],  
 # 'out\_works\_num': [],* **'in\_works'**: [],  
 **'out\_works'**: [],  
 **'layer'**: [],  
 **'visited'**: []  
 }  
 self.first\_top = **None** *# первая вершина графа* self.last\_top = **None** *# последняя вершина графа* self.works\_num = 0 *# количество работ графа* self.struct\_graph\_works\_num = 0 *# количество работ упорядоченного графа* self.current\_way = [] *# текущий путь для вывода всех полных путей* **with** open(file\_in) **as** File:  
 reader = csv.reader(File, delimiter=**';'**)  
 **for** row **in** reader:  
 self.add\_row\_in\_graph\_table(int(row[0]), int(row[1]), int(row[2]))  
 print(**'Чтение из файла проведено успешно'**)  
  
 **def** append\_new\_event\_in\_events\_graph(self, event):  
 *""" добавлние новой вершины в graph\_events, инит параметров нулями """* self.graph\_events[**'event'**].append(event)  
 self.graph\_events[**'in\_works'**].append(list())  
 self.graph\_events[**'out\_works'**].append(list())  
 self.graph\_events[**'layer'**].append(**None**)  
 self.graph\_events[**'early\_term'**].append(**None**)  
 self.graph\_events[**'late\_term'**].append(**None**)  
 self.graph\_events[**'reserve\_time'**].append(**None**)  
 self.graph\_events[**'visited'**].append(**None**)  
  
 **def** swap\_events(self, i, j):  
 *""" поменять местами строки событий"""  
 # шифр события* buff = self.graph\_events[**'event'**][i]  
 self.graph\_events[**'event'**][i] = self.graph\_events[**'event'**][j]  
 self.graph\_events[**'event'**][j] = buff  
 *# входящие работы* buff = self.graph\_events[**'in\_works'**][i]  
 self.graph\_events[**'in\_works'**][i] = self.graph\_events[**'in\_works'**][j]  
 self.graph\_events[**'in\_works'**][j] = buff  
 *# выходящие работы* buff = self.graph\_events[**'out\_works'**][i]  
 self.graph\_events[**'out\_works'**][i] = self.graph\_events[**'out\_works'**][j]  
 self.graph\_events[**'out\_works'**][j] = buff  
 *# слои* buff = self.graph\_events[**'layer'**][i]  
 self.graph\_events[**'layer'**][i] = self.graph\_events[**'layer'**][j]  
 self.graph\_events[**'layer'**][j] = buff  
 *# ранний срок* buff = self.graph\_events[**'early\_term'**][i]  
 self.graph\_events[**'early\_term'**][i] = self.graph\_events[**'early\_term'**][j]  
 self.graph\_events[**'early\_term'**][j] = buff  
 *# поздний срок* buff = self.graph\_events[**'late\_term'**][i]  
 self.graph\_events[**'late\_term'**][i] = self.graph\_events[**'late\_term'**][j]  
 self.graph\_events[**'late\_term'**][j] = buff  
 *# резерв времени* buff = self.graph\_events[**'reserve\_time'**][i]  
 self.graph\_events[**'reserve\_time'**][i] = self.graph\_events[**'reserve\_time'**][j]  
 self.graph\_events[**'reserve\_time'**][j] = buff  
 *# посещенность* buff = self.graph\_events[**'visited'**][i]  
 self.graph\_events[**'visited'**][i] = self.graph\_events[**'visited'**][j]  
 self.graph\_events[**'visited'**][j] = buff  
  
 **def** sort\_events\_by\_layers(self):  
 *""" сортировка событий по слоям по возрастанию """* print(**'\_'** \* 100)  
 print(**'Сортировка по слоям'**)  
 n = len(self.graph\_events[**'event'**])  
 **for** i **in** range(n - 1):  
 **for** j **in** range(n - i - 1):  
 **if** self.graph\_events[**'layer'**][j] > self.graph\_events[**'layer'**][j + 1]:  
 self.swap\_events(j, j + 1)  
 print(**'\_'** \* 100)  
  
 **def** find\_early\_term\_for\_event(self, event):  
 *""" нахождение раннего срока события """* event\_index = self.find\_index\_by\_event(event)  
 *# если событие начальное, ранний срок = 0* **if** event == self.first\_top:  
 self.graph\_events[**'early\_term'**][event\_index] = 0  
 **return** *# нахождение максимального веса из работ, связанных с предыдущими событиями* max\_early\_term = 0  
 tmp\_event\_early\_term = 0  
 *# ищем работу, у которой конечное событие равно нашему событию* **for** i **in** range(0, self.struct\_graph\_works\_num):  
 **if** event == self.struct\_graph\_table[**'arc\_end'**][i]:  
 max\_weight = self.struct\_graph\_table[**'weight'**][i]  
 prev\_event = self.struct\_graph\_table[**'arc\_start'**][i]  
 prev\_event\_index = self.find\_index\_by\_event(prev\_event)  
 prev\_event\_early\_term = self.graph\_events[**'early\_term'**][prev\_event\_index]  
 tmp\_event\_early\_term = prev\_event\_early\_term + max\_weight  
 **if** max\_early\_term < tmp\_event\_early\_term:  
 max\_early\_term = tmp\_event\_early\_term  
 self.graph\_events[**'early\_term'**][event\_index] = max\_early\_term  
  
 **def** find\_early\_term\_for\_all\_event(self):  
 *""" нахождение раннего срока всех событий """* print(**'\_'** \* 100)  
 print(**'Нахождение ранних сроков у событий'**)  
 **for** event **in** self.graph\_events[**'event'**]:  
 self.find\_early\_term\_for\_event(event)  
 print(**'\_'** \* 100)  
  
 **def** find\_late\_term\_for\_event(self, event):  
 *""" нахождение позднего срока события """* event\_index = self.find\_index\_by\_event(event)  
 *# если событие конечное, поздний срок = раннему сроку события* **if** event == self.last\_top:  
 self.graph\_events[**'late\_term'**][event\_index] = self.graph\_events[**'early\_term'**][event\_index]  
 **return** *# нахождение минимального веса из работ, связанных с последующими событиями* min\_late\_term = **None  
 for** i **in** range(self.struct\_graph\_works\_num):  
 **if** event == self.struct\_graph\_table[**'arc\_start'**][i]:  
 min\_weight = self.struct\_graph\_table[**'weight'**][i]  
 next\_event = self.struct\_graph\_table[**'arc\_end'**][i]  
 next\_event\_index = self.find\_index\_by\_event(next\_event)  
 next\_event\_late\_term = self.graph\_events[**'late\_term'**][next\_event\_index]  
 tmp\_event\_late\_term = next\_event\_late\_term - min\_weight  
  
 **if** min\_late\_term **is None**:  
 min\_late\_term = tmp\_event\_late\_term  
 **if** min\_late\_term > tmp\_event\_late\_term:  
 min\_late\_term = tmp\_event\_late\_term  
 self.graph\_events[**'late\_term'**][event\_index] = min\_late\_term  
  
 **def** find\_late\_term\_for\_all\_event(self):  
 *""" нахождение позднего срока всех событий """* print(**'\_'** \* 100)  
 print(**'Нахождение поздних сроков у событий'**)  
 **for** \_i **in** range(len(self.graph\_events[**'event'**]) - 1, -1, -1):  
 event = self.graph\_events[**'event'**][\_i]  
 self.find\_late\_term\_for\_event(event)  
 print(**'\_'** \* 100)  
  
 **def** find\_time\_reserves\_for\_all\_event(self):  
 *""" нахождение резервов времени для всех событий """* print(**'\_'** \* 100)  
 print(**'Нахождение резервов времени для событий'**)  
 **for** event **in** self.graph\_events[**'event'**]:  
 ind = self.find\_index\_by\_event(event)  
 reserve\_time = self.graph\_events[**'late\_term'**][ind] - self.graph\_events[**'early\_term'**][ind]  
 self.graph\_events[**'reserve\_time'**][ind] = reserve\_time  
 print(**'\_'** \* 100)  
  
 **def** get\_critical\_path\_length(self):  
 *""" получение критического пути из таблицы событий """* print(**'\_'** \* 100)  
 n = len(self.graph\_events[**'event'**])  
 \_critical\_path\_length = self.graph\_events[**'late\_term'**][n - 1]  
 print(**f'Критический путь равен: {**\_critical\_path\_length**}'**)  
 print(**'\_'** \* 100)  
 **return** \_critical\_path\_length  
  
 **def** find\_time\_reserve\_for\_work(self, index, full=**True**):  
 *""" нахождение полного или независмого временного резерва для работы """  
 # ПОЛНЫЙ ВРЕМЕННОЙ РЕЗЕРВ = раннее время конечного события работы - позднее время начального события работы -  
 # - время выполнения работы (вес работы)  
 # НЕЗАВИСИМЫЙ ВРЕМЕННОЙ РЕЗЕРВ = раннее время начального события работы -  
 # - позднее время конечного события работы - время выполнения работы (вес работы)  
 # full = True, то находится полный временной резерв* start\_work\_vertex = self.struct\_graph\_table[**'arc\_start'**][index] *# начальная вершина работы* end\_work\_vertex = self.struct\_graph\_table[**'arc\_end'**][index] *# конечная вершина работы* work\_weight = self.struct\_graph\_table[**'weight'**][index] *# вес работы* start\_work\_vertex\_index = self.find\_index\_by\_event(start\_work\_vertex) *# индекс нач вершины в таблице событий* end\_work\_vertex\_index = self.find\_index\_by\_event(end\_work\_vertex) *# индекс конеч вершины в таблице событий* **if** full:  
 end\_vertex\_late\_term = self.graph\_events[**'late\_term'**][end\_work\_vertex\_index] *# позднее время конеч события* start\_vertex\_early\_term = self.graph\_events[**'early\_term'**][start\_work\_vertex\_index] *# раннее время нач соб* full\_time\_reserve = end\_vertex\_late\_term - start\_vertex\_early\_term - work\_weight *# tп(j)-tр(i)-T(i,j)* self.struct\_graph\_table[**'full\_time\_reserve'**][index] = full\_time\_reserve  
 **else**:  
 start\_vertex\_late\_term = self.graph\_events[**'late\_term'**][start\_work\_vertex\_index] *# позднее время нач соб* end\_vertex\_early\_term = self.graph\_events[**'early\_term'**][end\_work\_vertex\_index] *# раннее время нач события* ind\_time\_reserve = end\_vertex\_early\_term - start\_vertex\_late\_term - work\_weight *# tр(j)-tп(i)-T(i,j)* self.struct\_graph\_table[**'independent\_time\_reserve'**][index] = ind\_time\_reserve  
  
 **def** find\_time\_reserves\_for\_all\_works(self):  
 *""" нахождение полных и независимых временных резервов для всех работ """* print(**'\_'** \* 100)  
 print(**f'Нахождение полных и независимых временных резервов для всех работ'**)  
 \_n = len(self.struct\_graph\_table[**'arc\_start'**])  
 **for** i **in** range(\_n):  
 self.find\_time\_reserve\_for\_work(i, full=**True**) *# подчет полного резерва* self.find\_time\_reserve\_for\_work(i, full=**False**) *# подсчет независимого резерва* print(**'\_'** \* 100)  
  
 **def** find\_index\_by\_event(self, event):  
 *""" нахождение номера (индекса) события в структуре по его шифру """* events\_len = len(self.graph\_events[**'event'**])  
 **for** i **in** range(events\_len):  
 **if** self.graph\_events[**'event'**][i] == event:  
 **return** i  
 **return None  
  
 def** check\_vertex\_includes\_only\_prev\_layers\_vertices(self, index):  
 *""" проверить, что в вершину входят только вершины предыдущих слоев """* **for** in\_work **in** self.graph\_events[**'in\_works'**][index]:  
 in\_work\_index = self.find\_index\_by\_event(in\_work)  
 **if** self.graph\_events[**'layer'**][in\_work\_index] **is None**:  
 **return False  
 return True  
  
 def** get\_max\_layer\_form\_prev\_vertices(self, index):  
 *""" получаем максимальный слой предыдущих вершин данной вершины """* in\_works = self.graph\_events[**'in\_works'**][index]  
 max\_layer = 0  
 **for** in\_work **in** in\_works:  
 vertex = self.find\_index\_by\_event(in\_work)  
 layer = self.graph\_events[**'layer'**][vertex]  
 **if** max\_layer < layer:  
 max\_layer = layer  
 **return** max\_layer  
  
 **def** find\_vertex\_layers(self):  
 *""" находим слой каждой вершины обходом в ширину"""  
 # если нет входящих работ - слой 0* start\_vertex = self.first\_top *# начальная вершина* end\_vertex = self.last\_top *# конечная вершина* queue = [start\_vertex]  
 start\_vertex\_index = self.find\_index\_by\_event(start\_vertex)  
 self.graph\_events[**'layer'**][start\_vertex\_index] = 0 *# нулевой слой для начальной вершины* self.graph\_events[**'visited'**][start\_vertex\_index] = **True** *# посещенность вершины* layer = 0  
 **while** len(queue) > 0:  
 layer += 1  
 *# удаляем первый (верхний элемент из очереди)* vertex = queue.pop(0)  
  
 vertex\_index = self.find\_index\_by\_event(vertex)  
 outs\_len = len(self.graph\_events[**'out\_works'**][vertex\_index]) *# количество выходящих вершин* **for** \_i **in** range(outs\_len):  
 out\_event = self.graph\_events[**'out\_works'**][vertex\_index][\_i] *# шифр соб., выходящего из текущ. соб.* out\_event\_index = self.find\_index\_by\_event(out\_event) *# индекс соб., выходящего из текущ. соб.* flag = self.check\_vertex\_includes\_only\_prev\_layers\_vertices(out\_event\_index)  
 *# flag = True, когда все входящие в текущую вершину вершины пройдены* **if** self.graph\_events[**'visited'**][out\_event\_index] **is not True and** flag:  
 queue.append(out\_event)  
 self.graph\_events[**'visited'**][out\_event\_index] = **True** prev\_max\_layer = self.get\_max\_layer\_form\_prev\_vertices(out\_event\_index)  
 self.graph\_events[**'layer'**][out\_event\_index] = prev\_max\_layer + 1  
 **if** out\_event\_index == end\_vertex:  
 **return True** *# если конца не обнаружено* **return False  
  
 def** find\_ins\_and\_outs(self):  
 *""" находим входящие и исходящие работы у вершин """* vertices\_number = len(self.graph\_events[**'event'**])  
 **for** \_i **in** range(0, vertices\_number):  
 **for** j **in** range(0, self.works\_num):  
 *# находим и добавляем структуру входящие вершины* **if** self.struct\_graph\_table[**'arc\_end'**][j] == self.graph\_events[**'event'**][\_i]:  
 self.graph\_events[**'in\_works'**][\_i].append(self.struct\_graph\_table[**'arc\_start'**][j])  
 *# находим и добавляем исходящие* **if** self.struct\_graph\_table[**'arc\_start'**][j] == self.graph\_events[**'event'**][\_i]:  
 self.graph\_events[**'out\_works'**][\_i].append(self.struct\_graph\_table[**'arc\_end'**][j])  
  
 **def** find\_all\_vertices(self):  
 *""" нахождение всех вершин графа """* print(**'Находим все вершины (события):'**)  
 self.current\_way = [self.first\_top]  
 self.append\_new\_event\_in\_events\_graph(self.first\_top)  
  
 **for** \_i **in** range(0, self.works\_num):  
 **if** self.graph\_table[**'arc\_start'**][\_i] == self.first\_top:  
 self.current\_way.append(self.graph\_table[**'arc\_end'**][\_i]) *# запомнить следуюущую вершину в списке путей* **if** self.graph\_table[**'arc\_end'**][\_i] **not in** self.graph\_events[**'event'**]:  
 self.append\_new\_event\_in\_events\_graph(self.graph\_table[**'arc\_end'**][\_i])  
 self.recursive\_find\_all\_vertices()  
  
 **def** recursive\_find\_all\_vertices(self):  
 *""" рекурсивный обход графа """* **if** self.current\_way[len(self.current\_way) - 1] == self.last\_top:  
 self.current\_way.pop() *# удаление последней вершины, возврат по стеку* **return  
  
 for** i **in** range(0, self.works\_num):  
 **if** self.graph\_table[**'arc\_start'**][i] == self.current\_way[len(self.current\_way) - 1]:  
 self.current\_way.append(self.graph\_table[**'arc\_end'**][i]) *# запомнить вершину* **if** self.graph\_table[**'arc\_end'**][i] **not in** self.graph\_events[**'event'**]:  
 self.append\_new\_event\_in\_events\_graph(self.graph\_table[**'arc\_end'**][i])  
 self.recursive\_find\_all\_vertices()  
 self.current\_way.pop() *# удаление последней вершины* **return  
  
 def** add\_row\_in\_graph\_table(self, start, end, weight, is\_visited=**False**, adding\_is\_in\_sort=**False**):  
 *""" добавить строку в таблицу графа """* **if** adding\_is\_in\_sort:  
 self.struct\_graph\_table[**'arc\_start'**].append(start)  
 self.struct\_graph\_table[**'arc\_end'**].append(end)  
 self.struct\_graph\_table[**'weight'**].append(weight)  
 self.struct\_graph\_table[**'is\_visited'**].append(is\_visited)  
 self.struct\_graph\_table[**'full\_time\_reserve'**].append(**None**)  
 self.struct\_graph\_table[**'independent\_time\_reserve'**].append(**None**)  
 self.struct\_graph\_works\_num += 1  
 **else**:  
 self.graph\_table[**'arc\_start'**].append(start)  
 self.graph\_table[**'arc\_end'**].append(end)  
 self.graph\_table[**'weight'**].append(weight)  
 self.graph\_table[**'is\_visited'**].append(is\_visited)  
 self.works\_num += 1  
  
 **def** copy\_row\_to\_struct\_graph\_table(self, index):  
 *""" копировать строку из неструктурированного графа в структурированный по индексу """* start = self.graph\_table[**'arc\_start'**][index]  
 end = self.graph\_table[**'arc\_end'**][index]  
 weight = self.graph\_table[**'weight'**][index]  
 is\_visited = self.graph\_table[**'is\_visited'**][index]  
 self.add\_row\_in\_graph\_table(start, end, weight, is\_visited, adding\_is\_in\_sort=**True**)  
  
 **def** delete\_row\_from\_graph\_table(self, index):  
 *""" удалить строку неструктрированного графа по индексу """* self.graph\_table[**'arc\_start'**].pop(index)  
 self.graph\_table[**'arc\_end'**].pop(index)  
 self.graph\_table[**'weight'**].pop(index)  
 self.graph\_table[**'is\_visited'**].pop(index)  
 self.works\_num -= 1  
  
 **def** print\_row(self, row\_index, sorted\_graph=**False**):  
 *""" распечатать строку соотвествующего графа """* **if** sorted\_graph:  
 print(**'|{start:^10}|{end:^10}|{weight:^25}|{full\_res:^25}|{ind\_res:^25}|'**.format(  
 start=self.struct\_graph\_table[**'arc\_start'**][row\_index],  
 end=self.struct\_graph\_table[**'arc\_end'**][row\_index],  
 weight=self.struct\_graph\_table[**'weight'**][row\_index],  
 full\_res=str(self.struct\_graph\_table[**'full\_time\_reserve'**][row\_index]),  
 ind\_res=str(self.struct\_graph\_table[**'independent\_time\_reserve'**][row\_index]),  
 ))  
 **else**:  
 print(**'|{start:^10}|{end:^10}|{weight:^10}|'**.format(  
 start=self.graph\_table[**'arc\_start'**][row\_index],  
 end=self.graph\_table[**'arc\_end'**][row\_index],  
 weight=self.graph\_table[**'weight'**][row\_index]))  
  
 **def** print\_graph(self, sorted\_graph=**False**):  
 *""" печать таблицы |A|B|time|"""* print(**'\_'** \* 100)  
 print(**'Вывод таблицы путей графа (работ)'**)  
 **if** sorted\_graph:  
 print(**'|{start:^10}|{end:^10}|{weight:^25}|{full\_res:^25}|{ind\_res:^25}|'**.format(  
 start=**'A'**,  
 end=**'B'**,  
 weight=**'Продолжительность (τ)'**,  
 full\_res=**'Полный резерв (Rп)'**,  
 ind\_res=**'Независимый резерв (Rн)'**,  
 ))  
 **else**:  
 print(**'|{start:^10}|{end:^10}|{weight:^10}|'**.format(  
 start=**'A'**,  
 end=**'B'**,  
 weight=**'Вес'**,  
 ))  
 **for** row\_index **in** range(0, self.works\_num):  
 self.print\_row(row\_index, sorted\_graph)  
 print(**f'Первая вершина: {**self.first\_top**}'**)  
 print(**f'Последняя вершина: {**self.last\_top**}'**)  
 print(**'\_'** \* 100)  
  
 **def** print\_events\_table(self):  
 print(**f'\nВывод таблицы вершин (событий)'**)  
 print(**'|{event:^20}|{ins:^20}|{outs:^20}|{layer:^10}|{early\_term:^25}|{late\_term:^25}|{reserve:^25}|'**.format(  
 event=**'Событие'**,  
 ins=**'Входящие события'**,  
 outs=**'Выходящие события'**,  
 layer=**'Слой'**,  
 early\_term=**'Ранний срок (t\_р)'**,  
 late\_term=**'Поздний срок (t\_п)'**,  
 reserve=**'Резерв времени (R(i))'**,  
 ))  
 n = len(graph.graph\_events[**'event'**])  
 **for** \_i **in** range(0, n):  
 print(  
 **'|{event:^20}|{ins:^20}|{outs:^20}|{layer:^10}|{early\_term:^25}|{late\_term:^25}|{reserve:^25}|'**.format(  
 event=self.graph\_events[**'event'**][\_i],  
 ins=str(self.graph\_events[**"in\_works"**][\_i]),  
 outs=str(self.graph\_events[**'out\_works'**][\_i]),  
 layer=self.graph\_events[**'layer'**][\_i],  
 early\_term=str(self.graph\_events[**'early\_term'**][\_i]),  
 late\_term=str(self.graph\_events[**'late\_term'**][\_i]),  
 reserve=str(self.graph\_events[**'reserve\_time'**][\_i]),  
 ))  
  
 **def** search\_first\_top(self):  
 *""" поиск первой вершины графа """* print(**'нахождение первой вершины'**)  
 **for** i **in** range(0, self.works\_num): *# итерация 1 по таблице* is\_start\_top = **True  
 for** j **in** range(0, self.works\_num): *# итерация 2 по таблице для поиска первой вершины  
 # условие, что в вершину входит дуга (кроме петель), если входит, то это не стартовая вершина* **if** (self.graph\_table[**'arc\_start'**][i] == self.graph\_table[**'arc\_end'**][j] **and** self.graph\_table[**'arc\_start'**][i] != self.graph\_table[**'arc\_start'**][j]):  
 is\_start\_top = **False  
 break** *# если это непосещенная первая вершина* **if** is\_start\_top **and not** self.graph\_table[**'is\_visited'**][i]:  
 *# если это первая первая вершина (не изменялась после инициализации)* **if** self.first\_top **is None**:  
 self.first\_top = self.graph\_table[**'arc\_start'**][i]  
 *# иначе она уже не первая, нарушаются правила СГ, нужно создавать фиктив. вершину* **else**:  
 print(**'создание фиктивной первой вершины'**)  
 self.add\_row\_in\_graph\_table(self.fictive\_start\_top, self.first\_top, 0)  
 self.add\_row\_in\_graph\_table(self.fictive\_start\_top, self.graph\_table[**'arc\_start'**][i], 0)  
 **if** self.first\_top != self.fictive\_start\_top:  
 self.first\_top = self.fictive\_start\_top  
  
 **for** index **in** range(0, self.works\_num):  
 **if** self.graph\_table[**'arc\_start'**][index] == self.graph\_table[**'arc\_start'**][i]:  
 self.graph\_table[**'is\_visited'**][index] = **True  
  
 def** delete\_top\_loops(self, index):  
 *""" удаление петель """* start\_top = self.graph\_table[**'arc\_start'**][index]  
 end\_top = self.graph\_table[**'arc\_end'**][index]  
 **if** start\_top == end\_top:  
 print(**f'найдена петля ({**start\_top**}, {**end\_top**})! автоматическое удаление работы'**)  
 self.delete\_row\_from\_graph\_table(index)  
 **return** 0  
 **return** index  
  
 **def** check\_duplication(self, i, j):  
 *""" проверка дубликатов работ """* start\_work = self.graph\_table[**'arc\_start'**][i]  
 starts\_are\_equal = start\_work == self.graph\_table[**'arc\_start'**][j]  
 end\_work = self.graph\_table[**'arc\_end'**][i]  
 ends\_are\_equal = end\_work == self.graph\_table[**'arc\_end'**][j]  
 *# если есть дубликаты* **if** starts\_are\_equal **and** ends\_are\_equal **and** i != j:  
 print(**f'ошибка! работа ({**start\_work**}, {**end\_work**}) дублируется'**)  
 *# если веса разные* **if** self.graph\_table[**'weight'**][i] != self.graph\_table[**'weight'**][j]:  
 weight\_i = self.graph\_table[**'weight'**][i]  
 weight\_j = self.graph\_table[**'weight'**][j]  
 print(**f'работа имеет 2 веса: {**weight\_i**} и {**weight\_j**}'**)  
 choice = input(**'Введите нужный вариант (1 или 2):'**)  
 **if** choice == **'1'**:  
 self.delete\_row\_from\_graph\_table(i)  
 self.optimize\_graph()  
 **return None  
 else**:  
 self.delete\_row\_from\_graph\_table(j)  
 **return** 0  
 *# если веса одинаковые* **else**:  
 print(**f'работа имеет одинаковый вес {**self.graph\_table[**"weight"**][i]**}; автоматическое удаление'**)  
 self.delete\_row\_from\_graph\_table(j)  
 **return** 0  
  
 **def** optimize\_graph(self):  
 *""" оптимизация графа (удаление петель, одинаковых работ, поиск последней вершины"""* print(**'оптимизация:'**)  
 i = 0  
 **while** i < self.works\_num:  
 *# one* i = self.delete\_top\_loops(i)  
 *# two* j = 0  
 **while** j < self.works\_num: *# сравнение всех вершин СГ* temp = self.check\_duplication(i, j)  
 *# None при запуске optimize\_graph рекурсивно из check\_duplication(i, j). Иначе обнуляется счётчик цикла.* j += 1  
 **if** temp **is not None**:  
 j = temp  
 i += 1  
 self.print\_graph()  
 self.search\_last\_top()  
 **for** index **in** range(0, self.works\_num):  
 self.graph\_table[**'is\_visited'**][index] = **False  
  
 def** search\_last\_top(self):  
 *""" поиск последней вершины графа """* print(**'нахождение последней вершины'**)  
 **for** i **in** range(0, self.works\_num): *# итерация 1 по таблице* is\_last\_top = **True  
 for** j **in** range(0, self.works\_num): *# итерация 2 по таблице для поиска последней вершины  
 # условие, что из вершины выходит дуга (кроме петель), если выходит, то это не последняя вершина* **if** (self.graph\_table[**'arc\_end'**][i] == self.graph\_table[**'arc\_start'**][j] **and** self.graph\_table[**'arc\_end'**][i] != self.graph\_table[**'arc\_end'**][j]):  
 is\_last\_top = **False  
 break** *# если это непосещенная последняя вершина* **if** is\_last\_top **and not** self.graph\_table[**'is\_visited'**][i]:  
 *# если это первая последняя вершина (не изменялась после инициализации)* **if** self.last\_top **is None**:  
 self.last\_top = self.graph\_table[**'arc\_end'**][i]  
 *# отметка всех работ с таким шифром как посещенные* **for** k **in** range(0, self.works\_num):  
 **if** self.graph\_table[**'arc\_end'**][k] == self.graph\_table[**'arc\_end'**][i]:  
 self.graph\_table[**'is\_visited'**][k] = **True  
 else**:  
 *# если это не одна и та же вершина* **if** self.last\_top != self.graph\_table[**'arc\_end'**][i]:  
 print(**f'вершины графа {**self.last\_top**} и {**self.graph\_table[**"arc\_end"**][i]**} конечные'**)  
 choice = input(**'Ввести фиктивную конечную вершину (иначе удалить) (1-да, 2-нет): '**)  
 **if** choice == **'1'**:  
 print(**'создание фиктвной последней вершины'**)  
 self.add\_row\_in\_graph\_table(self.last\_top, self.fictive\_end\_top, 0)  
 self.add\_row\_in\_graph\_table(self.graph\_table[**'arc\_end'**][i], self.fictive\_end\_top, 0)  
 **if** self.last\_top != self.fictive\_end\_top:  
 self.last\_top = self.fictive\_end\_top  
  
 **for** index **in** range(0, self.works\_num):  
 **if** self.graph\_table[**'arc\_end'**][index] == self.graph\_table[**'arc\_end'**][i]:  
 self.graph\_table[**'is\_visited'**][index] = **True** *# удаление одной из вершин* **else**:  
 print(**f'удаление вершины {**self.graph\_table[**"arc\_end"**][i]**}'**)  
 self.delete\_row\_from\_graph\_table(i)  
 self.last\_top = **None  
 for** index **in** range(0, self.works\_num):  
 self.graph\_table[**'is\_visited'**][index] = **False** self.search\_last\_top()  
 **break  
  
 def** struct\_graph(self):  
 print(**'упорядочивание графа'**)  
 vertex\_queue = [self.first\_top]  
 **while** self.works\_num != self.struct\_graph\_works\_num:  
 **for** i **in** range(0, self.works\_num):  
 **if** vertex\_queue[0] == self.graph\_table[**'arc\_start'**][i] **and not** self.graph\_table[**'is\_visited'**][i]:  
 self.copy\_row\_to\_struct\_graph\_table(i)  
 vertex\_queue.append(self.graph\_table[**'arc\_end'**][i])  
 self.graph\_table[**'is\_visited'**][i] = **True** vertex\_queue.pop(0)  
  
 **def** search\_full\_ways(self, check\_critical=**False**):  
 *""" нахождение всех полных путей графа """* self.current\_way = [self.first\_top]  
 print(**'полные пути графа:'**)  
 **for** i **in** range(0, self.works\_num):  
 **if** self.graph\_table[**'arc\_start'**][i] == self.first\_top:  
 self.current\_way.append(self.graph\_table[**'arc\_end'**][i]) *# запомнить следуюущую вершину в списке путей* self.search\_bfs(check\_critical)  
 print(**'\_'** \* 100)  
  
 **def** check\_way\_criticality(self):  
 *""" проверка пути на то, что он является критичным """* **for** \_i **in** range(1, len(self.current\_way)):  
 prev\_way\_vertex = self.current\_way[\_i - 1]  
 way\_vertex = self.current\_way[\_i]  
 **for** \_j **in** range(self.struct\_graph\_works\_num):  
 *# если определенная работа и пути найдена* start\_work\_vertex = self.struct\_graph\_table[**'arc\_start'**][\_j]  
 end\_work\_vertex = self.struct\_graph\_table[**'arc\_end'**][\_j]  
 **if** start\_work\_vertex == prev\_way\_vertex **and** end\_work\_vertex == way\_vertex:  
 *# если у какого-то события в этом пути Rп != 0 (событие не критическое), то путь не критический* **if** self.struct\_graph\_table[**'full\_time\_reserve'**][\_j] != 0:  
 **return False** *# все события в пути критические, то путь критический* **return True  
  
 def** search\_bfs(self, check\_critical):  
 *""" обход графа в ширину"""* **if** self.current\_way[len(self.current\_way) - 1] == self.last\_top:  
 **if** check\_critical:  
 way\_is\_critical = self.check\_way\_criticality()  
 **if** way\_is\_critical:  
 print(**'Критический путь:'**)  
 **for** \_i **in** range(0, len(self.current\_way)):  
 print(**f'{**self.current\_way[\_i]**} '**, end=**''**)  
 print()  
 **else**:  
 **for** \_i **in** range(0, len(self.current\_way)):  
 print(**f'{**self.current\_way[\_i]**} '**, end=**''**)  
 print()  
 self.current\_way.pop() *# удаление последней вершины, возврат по стеку* **return  
  
 for** \_i **in** range(0, self.works\_num):  
 **if** self.graph\_table[**'arc\_start'**][\_i] == self.current\_way[len(self.current\_way) - 1]:  
 self.current\_way.append(self.graph\_table[**'arc\_end'**][\_i]) *# запомнить вершину* self.search\_bfs(check\_critical)  
 self.current\_way.pop() *# удаление последней вершины* **return  
  
  
if** \_\_name\_\_ == **"\_\_main\_\_"**:  
 in\_file = **"input.csv"** graph = Graph(in\_file)  
 graph.print\_graph()  
 *# находим первую вершину* graph.search\_first\_top()  
 *# оптимизируем (удаляем петли, дубли), там же находим конечную вершину* graph.optimize\_graph()  
 graph.print\_graph()  
 *# структурируем граф* graph.struct\_graph()  
 graph.print\_graph(sorted\_graph=**True**)  
 *# находим полные пути* graph.search\_full\_ways()  
 *# находим все вершины* graph.find\_all\_vertices()  
 *# находим входящие и исходящие вершины* graph.find\_ins\_and\_outs()  
 *# находим слои у вершин* graph.find\_vertex\_layers()  
 graph.print\_events\_table()  
 *# сортируем таблицу событий по слоям* graph.sort\_events\_by\_layers()  
 graph.print\_events\_table()  
 *# находим ранние сроки для событий* graph.find\_early\_term\_for\_all\_event()  
 *# находим поздние сроки для событий* graph.find\_late\_term\_for\_all\_event()  
 *# находим временной резерв для событий* graph.find\_time\_reserves\_for\_all\_event()  
 graph.print\_events\_table()  
 *# находим полные и независимые временные резервы для работы* graph.find\_time\_reserves\_for\_all\_works()  
 graph.print\_graph(sorted\_graph=**True**)  
 *# выводим критические пути* graph.search\_full\_ways(check\_critical=**True**)  
 *# находим длину критического пути* critical\_path\_length = graph.get\_critical\_path\_length()