Министерство образования и науки Российской Федерации

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ
ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
«САРАТОВСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ
ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ ИМЕНИ Н. Г. ЧЕРНЫШЕВСКОГО»

Кафедра информатики и программирования

Реализация алгоритма Флойда поиска кратчайших путей на графе с использованием технологий ASP.NET Web API и Windows Forms

КУРСОВАЯ РАБОТА

Студента 3 курса 341 группы
направления 02.03.03 – Математическое обеспечение и администрирование
информационных систем (МОиАИС)
факультета компьютерных наук и информационных технологий (КНиИТ)
Акимова Артемия Андреевича

Научный руководитель
Доцент кафедры ИиП, к. ф.-м. н. _____ К. П. Вахлаева
подпись, дата

Консультант
Глава лаборатории .NET учебного центра
саратовского филиала компании
«EPAM Systems» _____ Д. М. Вересkун
подпись, дата

Зав. кафедрой
Доцент кафедры ИиП, к. ф.-м. н. _____ А. Г. Федорова
подпись, дата

Саратов 2016

Содержание

Введение

Поиск кратчайшего пути – важная задача, возникающая в разных областях науки и техники: в экономике (при оптимизации перевозок), в робототехнике (при поиске роботом оптимального маршрута), в компьютерных играх. На этапе моделирования традиционно используются графы, вершины которых соответствуют пунктам назначения, а ребра – прямым маршрутам из одного пункта в другой. Часто ребру ставится в соответствие число, характеризующее условную «стоимость» перемещения.

Приложение по поиску таких путей имеет большой спектр использования, но есть проблема – как предоставить это приложение клиентам по всему миру. Для того чтобы предоставить такую услугу удаленно необходимо разделить приложения на две части – клиентскую и серверную.

Целью курсовой работы являлось изучение технологий работы с клиент-серверным приложением, алгоритма поиска кратчайших путей на большом разреженном графе и наглядная визуализация хода его работы.

Данная цель предполагает решение следующих основных ***задач***:

1. изучение алгоритма Флойда поиска кратчайших путей на графе и его реализация на языке программирования C#;
2. изучение технологии ASP.NET Web API;
3. реализация Windows Forms приложения с использованием ASP.NET Web API;
4. тестирование реализованного приложения на наборе тестов для проверки алгоритмов поиска кратчайших путей.

1 Поиск кратчайших путей для всех пар вершин на графе

1.1 Основные определения и обозначения

Граф – это пара $G=(V,E)$, где V - множество вершин, E - множество ребер.

Ориентированным называется граф, в котором $EV\times V$ – множество упорядоченных пар вершин вида (v_i,v_j), где v_i называется *началом*, а v_j – *концом* дуги.

Для любого графа $G=(V,E)$ полагаем, что множество вершин V имеет мощность: $|V|=n$, а множество ребер E имеет мощность: $|E|=m$.

Неориентированным называется граф, в котором $E\{(v_i,v_j):v_i,v_j\in V\wedge v_i\neq v_j\}$ – множество неупорядоченных пар вершин. Элементы этого множества называются ***ребрами***.

Вершины, соединенные ребром, называются ***смежными***. Ребра, имеющие общую вершину, также называются *смежными*. Ребро и любая из двух его вершин называются ***инцидентными***.

На практике неориентированный граф используется для задания симметричных отношений для объектов.

Граф называется ***взвешенным***, если каждой его дуге (ребру) поставлена в соответствие некоторая числовая характеристика $w_{ij}=w(v_i,v_j),v_i,v_j\in V$, называемая ***весом*** данной дуги. На рисунке 1 представлен пример взвешенного ориентированного графа.

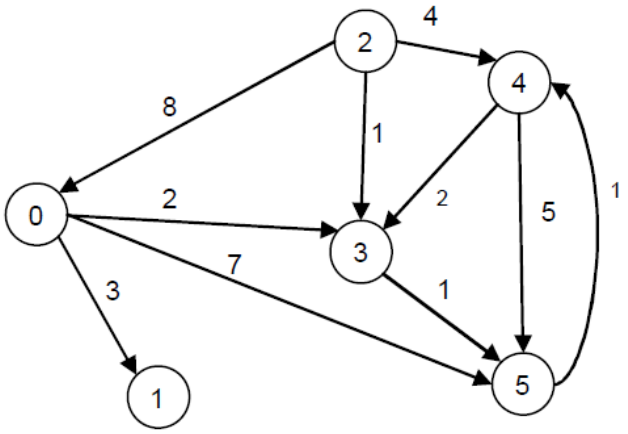

Рисунок 1. Взвешенный ориентированный граф

Путем, соединяющим вершины v_0, v_n, назовем такую последовательность вершин $v_0, v_1, \ldots, v_n$, $n \geq 0$, что для всех i, $0 \leq i < n-1$, существуют ребра (v_i, v_{i+1}) или дуги соответственно.

Длиной пути в графе полагают количество ребер (дуг), составляющих этот путь. ***Длиной пути во взвешенном графе*** полагают сумму весов всех дуг (ребер), входящих в этот путь.

1.2 Формат хранения графа

Графы для экспериментов могу быть получены с помощью специальных генераторов или на основании реальных данных, например, можно использовать граф сети дорог, содержащий расстояния или время между узловым точками. В качестве примера в данной работе выбран граф карты дорог Рима [4]. Граф карты дорого Рима задается в файле и имеет текстовый формат. Файл содержит строчки следующих типов:

1. Строка с описанием графа, например, "sp 2000 6000", означает, что граф разреженный и содержит 2000 вершин, 6000 ребер.
2. Список ребер графа, например, "596 959 78", означает ребро из вершины 596 в вершину 959 с весом 78.

Граф карты дорог Рима имеет разреженный формат, поэтому для его хранения в оперативной памяти компьютера будет использоваться ***строчный формат CRS*** (***Compressed Sparse Rows***) хранения разреженных матриц.

1.2.1 Хранение разреженной матрицы

Разреженная матрица — это матрица с преимущественно нулевыми элементами как показано, например, на рисунке 2.

$$\begin{pmatrix} 2 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 2 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 2 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 2 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 2 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 & 2 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 & 2 \end{pmatrix}$$

Рисунок 2. Разреженная матрица

Среди специалистов нет единства в определении того, какое именно количество ненулевых элементов делает матрицу разреженной. Разные авторы предлагают различные варианты. Огромные разреженные матрицы часто возникают при решении таких задач, как дифференциальные уравнения в частных производных.

Операции и алгоритмы, применяемые для работы с обычными, плотными матрицами, работают относительно медленно и требуют значительных объемов памяти, когда применяются к большим разреженным матрицам, поэтому при хранении и преобразовании разреженных матриц бывает полезно, а часто и необходимо, использовать специальные алгоритмы и структуры данных, которые учитывают разреженную структуру матрицы.

Один из наиболее простых для понимания форматов хранения разреженных матриц – ***координатный формат***. Элементы матрицы и ее структура хранятся в трех массивах, содержащих значения, номер строки и столбца, как показано на рисунке 3:

Рисунок 3. Координатный формат хранения

Координатный формат можно использовать как в упорядоченном виде, когда в массиве Value хранятся элементы матрицы построчно, например, слева направо и сверху вниз, так и в неупорядоченном варианте. Неупорядоченное представление существенно упрощает операции вставки/удаления новых элементов, но приводит к переборным поискам. Упорядоченный вариант позволяет быстрее находить все элементы нужной строки (столбца, если упорядочивать сначала по столбцам), но приводит к перепаковкам при вставках/удалениях элементов. В целом координатный формат достаточно прост, что является его несомненным достоинством, но оказывается недостаточно эффективным с точки зрения использования памяти и основных алгоритмов обработки.

Формат хранения, широко распространенный под названием ***CSR*** (***Compressed Sparse Rows***) или ***CRS*** (***Compressed Row Storage***), призван устранить некоторые недоработки координатного представления. В формате CSR также используются три массива. Первый массив хранит значения элементов построчно (строки рассматриваются по порядку сверху вниз), второй – номера столбцов для каждого элемента, а третий заменяет номера строк, используемые в координатном формате, на индекс начала каждой строки.

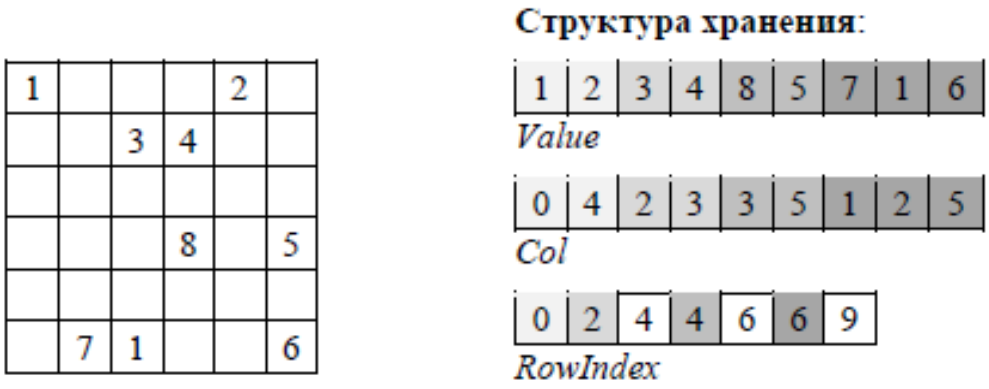

Рисунок 4. Разреженный строчный формат хранения

Отметим, что количество элементов массива $RowIndex$ равно $N+1$. При этом элементы строки i в массиве $Value$ находятся по индексам от $RowIndex[i]$ до $RowIndex[i+1]-1$ включительно (i-ый элемент массива $RowIndex$ указывает на начало i-ой строки). Исходя из этого обрабатывается случай пустых строк, а также добавляется «лишний» элемент в массив $RowIndex$ – устраняется особенность при доступе к элементам последней строки. Данный элемент хранит номер последнего ненулевого элемента матрицы плюс один, что соответствует количеству ненулевых элементов.

Формат CSR является более эффективным, чем координатный, с точки зрения объема используемой памяти.

Далее в работе будем использовать формат CRS в виде трех массивов, расположенных в памяти в порядке *Col*, *pointerB* (*RowIndex*) и *Value*.

1.3 Алгоритм Флойда

Пусть дан взвешенный граф $G=(V,E)$, содержащий n вершин, в котором каждому ребру приписан неотрицательный вес, а также в графе отсутствуют петли. ***Задача поиска кратчайших путей между всеми парами вершин*** состоит в том, что для имеющегося графа G требуется получить матрицу расстояний, где содержится минимальная длина пути от одной (произвольной) вершины графа до любой другой.

Существует несколько классических алгоритмов решения задачи поиска кратчайших путей. Можно решать задачу, последовательно применяя алгоритм Дейкстры для каждой вершины, объявляемой в качестве источника, но существуют также прямые методы решения, использующие алгоритмы Флойда или Джонсона. В данной работе рассмотрен алгоритм поиска кратчайших путей, предложенный Флойдом.

В основе алгоритма поиска кратчайших путей используется структура кратчайших путей. Рассмотрим путь $p=(v_0, v_1, \ldots, v_i, v_j, \ldots, v_{k-1})$. Пусть данный путь кратчайший, тогда пути $p_{0,i}=(v_0, \ldots, v_i)$, $p_{i,j}=(v_i, v_j)$, $p_{j,k-1}=(v_j, \ldots, v_{k-1})$ также будут кратчайшими.

Этот факт легко обосновать тем, что длина пути складывается из суммы длин его частей. Пусть к пути p необходимо добавить вершину v_k, тогда возможны два варианта:

1. Стоимость пути $p_{i,j}$ меньше стоимости пути $p'_{i,j}=(v_i, v_k, v_j)$, в этом случае кратчайший путь не изменится.
2. Стоимость пути $p_{i,j}$ больше стоимости пути $p'_{i,j}=(v_i, v_k, v_j)$, тогда кратчайшим будет путь $p'=(v_0, v_1, \ldots, v_i, v_k, v_j, \ldots, v_{k-1})$. Данный факт следует из того условия, что длина пути складывается из длины пути $p'_{i,j}$ и оставшихся частей.

Используя данные утверждения, запишем алгоритм Флойда.

Алгоритм Флойда:

1. Перенумеруем вершины исходного графа числами от 0 до $n-1$.
2. Определим матрицу $D=(d_{i,j})$ расстояний между вершинами без промежуточных вершин значения элементов которой $d_{i,j}$, $0 \le i, j \le n-1$

совпадают с весами $w_{i,j}$ перехода из вершины i в вершину j. Если ребро $e_{i,j}$ отсутствует, то $d_{i,j}=\infty$ кроме того $d_{i,i}=0$.

$$d_{i,j}=\begin{cases} w_{i,j}, если\ e_{i,j}\in E, \\ 0, если\ i=j, \\ \infty, иначе \end{cases}$$

3. Пусть $d_{i,j}^{k}$ – длина кратчайшего пути из вершины i в вершину j с проверенной возможностью прохождения через вершину k в качестве промежуточной. D^{k} – матрица размера $n\times n$, элемент (i,j) которой совпадает с $d_{i,j}^{k}$. Для каждой вершины k последовательно принимающей значения $0,1,\ldots,n-1$ определим по величинам элементов матрицы D^{k-1} элементы матрицы D^{k}, используя рекуррентное соотношение:

$$d_{i,j}^{k}=\begin{cases} min\left(d_{i,j}, d_{i,k}^{\square}+d_{k,j}^{\square}\right), если\ k=0, \\ ¿\, min\left(d_{i,j}^{k-1}, d_{i,k}^{k-1}+d_{k,j}^{k-1}\right), если\ k>0, \end{cases} \qquad (*)$$

Используя это соотношение, можно найти матрицу $D^{n-1}=\left(d_{i,j}^{n-1}\right)$, содержащую длины кратчайших путей для всех пар вершин $i,j\in V$.

Замечание 1. Для поиска кратчайших путей часто вычисляют *матрицу предшествования* P, используя простые рекуррентные соотношения:

$$P_{i,j}^{\square}=\begin{cases} NULL, если\ w_{i,j}=\infty, \\ i, если\ i=j, \\ i, если\ w_{i,j}<\infty. \end{cases}$$

$$P_{i,j}^{0}=\begin{cases} P_{i,j}^{\square}, если\ d_{i,j}^{\square}\le d_{i,k}^{\square}+d_{k,j}^{\square}, \\ ¿\, P_{k,j}^{\square}, если\ d_{i,j}^{\square}>d_{i,k}^{\square}+d_{k,j}^{\square}. \end{cases} \qquad (\ k=0\ )$$

$$P_{i,j}^{k}=\begin{cases} P_{i,j}^{k-1}, если\ d_{i,j}^{k-1}\le d_{i,k}^{k-1}+d_{k,j}^{k-1}, \\ ¿\, P_{k,j}^{k-1}, если\ d_{i,j}^{k-1}>d_{i,k}^{k-1}+d_{k,j}^{k-1}, \end{cases} \qquad (\ k>0\ )$$

Сложность алгоритма Флойда имеет порядок n^3, что означает при увеличении количества вершин в графе в 2 раза увеличение времени работы алгоритма в 8 раз и так далее.

Как можно заметить, в ходе выполнения алгоритма матрица кратчайших путей изменяется, после завершения вычисления в матрице будет храниться требуемый результат – длины линий кратчайших путей для каждой пары вершин исходного графа.

В качестве примера выполнения алгоритма Флойда рассмотрим граф, представленный на рисунке 5:

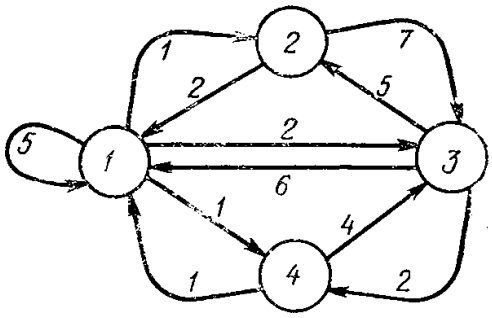

Рисунок 5. Взвешенный ориентированный граф

Для графа, изображенного на рисунке 5, матрица D^0, составленная из длин дуг графа такова:

$$D^0 = \begin{matrix} 0 & 1 & 2 & 1 \\ 2 & 0 & 7 & \infty \\ 6 & 5 & 0 & 2 \\ 1 & \infty & 4 & 0 \end{matrix}$$

Величины элементов матрицы D^1 и соответствующие им кратчайшие пути определяются следующим образом:

$d_{ij}^1 = min\left(d_{i1}^0 + d_{1j}^0, d_{ij}^0\right)$	Соответствующие пути
$d_{11}^1 = d_{11}^0 = 0$	
$d_{12}^1 = d_{12}^0 = 1$	(1,2)

$d_{13}^1 = d_{13}^0 = 2$	(1,3)
$d_{14}^1 = d_{14}^0 = 1$	(1,4)
$d_{21}^1 = d_{21}^0 = 2$	(2,1)
$d_{22}^1 = 0$	
$d_{23}^1 = min\left(d_{21}^0 + d_{13}^0, d_{23}^0\right) = min(2+2, 7) = 4$	(2,1), (1,3)
$d_{24}^1 = min\left(d_{21}^0 + d_{14}^0, d_{24}^0\right) = min(2+1, \infty) = 3$	(2,1), (1,4)
$d_{31}^1 = d_{31}^0 = 6$	(3,1)
$d_{32}^1 = min\left(d_{31}^0 + d_{12}^0, d_{32}^0\right) = min(6+1, 5) = 5$	(3,2)
$d_{33}^1 = 0$	
$d_{34}^1 = min\left(d_{31}^0 + d_{14}^0, d_{34}^0\right) = min(6+1, 2) = 2$	(3,4)
$d_{41}^1 = d_{41}^0 = 1$	(4,1)
$d_{42}^1 = min\left(d_{41}^0 + d_{12}^0, d_{42}^0\right) min(1+1, \infty) = 2$	(4,1), (1,2)
$d_{43}^1 = min\left(d_{41}^0 + d_{13}^0, d_{43}^0\right) min(1+2, 4) = 3$	(4,1), (1,3)
$d_{44}^1 = d_{44}^0 = 0$	

Аналогичным образом могут быть определены величины элементов матриц $D^2, D^3 u D^4$ и соответствующие им кратчайшие пути. Полученные результаты приводятся ниже.

Матрицы D^2 :

$$D^2 = \begin{bmatrix} 0 & 1 & 2 & 1 \\ 2 & 0 & 4 & 3 \\ 6 & 5 & 0 & 2 \\ 1 & 2 & 3 & 0 \end{bmatrix}$$

Кратчайшие пути для элементов матрицы D^2 :

$$\begin{bmatrix} & & ¿ \\ (1,2) & (1,3) & (1,4) \\ (2,1) & ¿(2,1),(1,3) & (2,1),(1,4) \\ (3,1) & (3,2) & ¿(3,4) \\ (4,1) & (4,1),(1,2) & (4,1),(1,3) \end{bmatrix}$$

Матрицы D^3 :

$$D^3=\begin{bmatrix} 0 & 1 & 2 & 1 \\ 2 & 0 & 4 & 3 \\ 6 & 5 & 0 & 2 \\ 1 & 2 & 3 & 0 \end{bmatrix}$$

Кратчайшие пути для элементов матрицы D^3 :

$$\begin{bmatrix} & ¿ & \\ (1,2) & (1,3) & (1,4) \\ (2,1) & ¿(2,1),(1,3) & (2,1),(1,4) \\ (3,1) & (3,2) & ¿(3,4) \\ (4,1) & (4,1),(1,2) & (4,1),(1,3) \end{bmatrix}$$

Матрицы D^4 :

$$D^4=\begin{bmatrix} 0 & 1 & 2 & 1 \\ 2 & 0 & 4 & 3 \\ 3 & 4 & 0 & 2 \\ 1 & 2 & 3 & 0 \end{bmatrix}$$

Кратчайшие пути для элементов матрицы D^4 :

$$\begin{bmatrix} & ¿ & \\ (1,2) & (1,3) & (1,4) \\ (2,1) & ¿(2,1),(1,3) & (2,1),(1,4) \\ (3,4),(4,1) & (3,4),(4,1),(1,2) & ¿(3,4) \\ (4,1) & (4,1),(1,2) & (4,1),(1,3) \end{bmatrix}$$

Программа, использующая алгоритм Флойда, содержит вложенный цикл, который может быть представлен в общем виде следующим образом:

```
for(k = 0; k < n; k++)
     for(i = 0; i < n; i++)
```

```
for(j = 0; j < n; j++)
     D[i,j] = min(D[i,j], D[i,k]+D[k,j]);
```

2 Разработка приложения для визуализации алгоритма на основе клиент-серверной архитектуры

2.1 Клиент-серверная архитектура

Клиент-сервер – вычислительная или сетевая архитектура, в которой задания или сетевая нагрузка распределены между поставщиками услуг, называемыми ***серверами***, и заказчиками услуг, называемыми ***клиентами***. Физически клиент и сервер – это программное обеспечение. Обычно они взаимодействуют через компьютерную сеть посредством сетевых протоколов и находятся на разных вычислительных машинах, но могут выполняться также и на одной машине. Программы-сервера, ожидают от клиентских программ запросы и предоставляют им свои ресурсы в виде данных, например, загрузка файлов посредством HTTP, FTP, BitTorrent, потоковое мультимедиа или работа с базами данных, или сервисных функций, например, работа с электронной почтой, общение посредством систем мгновенного обмена сообщениями, просмотр веб-страниц во Всемирной паутине.

Рисунок 6. Клиент-серверная архитектура

Клиент-серверная архитектура характеризуется описыванием отношений взаимодействующих программ в приложении. Серверный компонент обеспечивает функцию или услугу на один или многие клиенты, которые инициируют запросы на такие услуги.

Сервера подразделяются на услуги, которые они предоставляют. Например, веб-сервер, обслуживающий веб-страниц и файл-сервер обслуживает компьютерные файлы. Общий ресурс может быть одним из видов программного обеспечения сервера и электронных компонентов, от программ и данных к процессорам и запоминающим устройствам. Разделенный ресурс сервера представляет собой службу.

Является ли компьютер клиентом, сервером, или одновременно и тем, и другим, определяется характером приложения, которое использует сервисные функции. Например, на одном компьютере можно запустить веб-сервер и файл-сервер программного обеспечения, и в то же время может использоваться для обслуживания различных данных клиентов, оформляющих различные виды запросов. Клиентское программное обеспечение может соединяться с серверным программным обеспечением на одном компьютере.

2.1.1 Протоколы взаимодействия клиента и сервера

Существует несколько видов протоколов, но наиболее известными являются SOAP(Simple Object Access Protocol) и REST-сервисы. Ограничимся изучением REST-сервиса и его характерными отличиями от SOAP.

REST (**Representational State Transfer** – передача состояния представления) – архитектурный стиль взаимодействия компонентов распределенного приложения в сети. REST представляет собой согласованный набор ограничений, учитываемых при проектировании распределенной гипермедиа-системы. Такой подход приводит к повышению производительности и надежности системы, масштабируемости, гибкости, портативности компонентов, способности эволюционировать, приспосабливаясь к новым требованиям.

Для построения распределенных REST-приложений необходимо выполнение ряда условий по Филдингу[5].

1. **Клиент-серверная архитектура.** Единый интерфейс между клиентом и сервером. Такое разделение подразумевает отсутствие связи между клиентами и хранилищем данных. Это хранилище остаётся внутренним

устройством сервера, таким образом переносимость клиентского кода увеличивается, что способствует упрощению сервера и его масштабируемости. Серверы и клиенты могут быть мгновенно заменены независимо друг от друга, так как интерфейс между ними не меняется.

2. **Отсутствие состояния.** Серверы не связаны с интерфейсами клиентов и их состояниями. На стороне сервера не сохраняется пользовательский контекст между двумя разными запросами. Каждый запрос содержит всю информацию, необходимую обработчику, а состояние сессии хранится на клиенте. Состояние сессии может быть передано сервером на другой сервис благодаря поддержке постоянного состояния базой данных. Клиент отсылает запросы, когда готов совершить транзакцию на изменение состояния.
3. **Кэширование.** Как и во Всемирной паутине, каждый из клиентов, а также промежуточные узлы между сервером и клиентами могут кэшировать ответы сервера. В каждом запросе клиента должно явно содержаться указание о возможности кэширования ответа и получения ответа из существующего кэша. В свою очередь, ответы могут явно или неявно определяться как кэшируемые или некэшируемые для предотвращения повторного использования клиентами в последующих запросах сохранённой информации. Правильное использование кэширования в REST-архитектуре устраняет избыточные клиент-серверные взаимодействия, что улучшает скорость и расширяемость системы.
4. **Единообразие интерфейса.** Ограничения на унифицированный интерфейс являются фундаментальными в дизайне REST-сервисов. Каждый из сервисов функционирует и развивается независимо. Ограничения для унификации интерфейса:
    a. Идентификация ресурсов. Индивидуальные ресурсы идентифицированы в запросах, например, с использованием URI в интернет-системах. Ресурсы сами по себе отделены от представлений, которые возвращаются клиентам. Например, сервер может отсылать данные из

базы данных в виде HTML, XML или JSON, ни один из которых не является типом хранения внутри хранилища сервера.

b. Манипуляция ресурсами через представление. В момент, когда клиенты хранят представление ресурса, включая метаданные, они имеют достаточно данных для модификации или удаления ресурса.
c. «Самодостаточные» сообщения. Каждое сообщение достаточно информативно для того, чтобы описать каким образом его обрабатывать.
d. Гипермедиа, как средство изменения состояния сервера. Клиенты могут изменить состояние системы только через действия, которые динамически идентифицируются на сервере посредством гипермедиа (к примеру, гиперссылки в гипертексте, формы связи, флажки, радиокнопки и прочее).

5. **Слои.** Клиент может взаимодействовать не напрямую с сервером, а через промежуточные узлы (слои). При этом клиент может не знать об их существовании, за исключением случаев передачи конфиденциальной информации. Промежуточные серверы выполняют балансировку нагрузки и могут использовать дополнительное кэширование.
6. **Код по требованию (необязательное ограничение).** REST может позволить расширить функциональность клиента за счёт загрузки кода с сервера в виде апплетов или сценариев. Филдинг утверждает, что дополнительное ограничение позволяет проектировать архитектуру, поддерживающую желаемую функциональность в общем случае, но возможно за исключением некоторых контекстов.

Отличия REST-сервиса от SOAP

SOAP активно использует XML для кодирования запросов и ответов, а также строгую типизацию данных, гарантирующую их целостность при передаче между клиентом и сервером. С другой стороны, запросы и ответы в REST могут передаваться в ASCII, XML, JSON или любых других форматах, распознаваемых одновременно и клиентом, и сервером. Кроме того, в модели REST отсутствуют встроенные требования к типизации данных. В результате

пакеты запросов и ответов в REST имеют намного меньшие размеры, чем соответствующие им пакеты SOAP.

В модели SOAP уровень передачи данных протокола HTTP является «пассивным наблюдателем», и его роль ограничивается передачей запросов SOAP от клиента серверу с использованием метода POST. Детали сервисного запроса, такие как имя удаленной процедуры и входные аргументы, кодируются в теле запроса. Архитектура REST, напротив, рассматривает уровень передачи данных HTTP как активного участника взаимодействия, используя существующие методы HTTP, такие как GET, POST, PUT и DELETE, для обозначения типа запрашиваемого сервиса. Следовательно, с точки зрения разработчика, запросы REST в общем случае более просты для формулирования и понимания, так как они используют существующие и хорошо понятные интерфейсы HTTP.

Модель SOAP поддерживает определенную степень интроспекции, позволяя разработчикам сервиса описывать его API в файле формата Web Service Description Language (WSDL, язык описания веб-сервисов). Создавать эти файлы довольно сложно, однако это стоит затраченных усилий, поскольку клиенты SOAP могут автоматически получать из этих файлов подробную информацию об именах и сигнатурах методов, типах входных и выходных данных и возвращаемых значениях. С другой стороны, модель REST избегает сложностей WSDL в угоду более интуитивному интерфейсу, основанному на стандартных методах HTTP, описанных выше.

В основе REST лежит концепция ресурсов, в то время как SOAP использует интерфейсы, основанные на объектах и методах. Интерфейс SOAP может содержать практически неограниченное количество методов; интерфейс REST, напротив, ограничен четырьмя возможными операциями, соответствующими четырем методам HTTP.

2.2 Реализация REST-сервиса с помощью ASP.NET WEB API

Самым актуальный способом создать REST-сервис в стеке технологий Microsoft на сегодняшний день является ASP.NET Web API. ASP.NET Web API.

по сути позволяет выполнять CRUD (Create Read Update Delete) операции создания, чтения, обновления и удаления над ресурсами, используя HTTP. Для этой цели REST использует ограниченный набор глаголов HTTP запросов:

1. GET: используется для чтения или получения ресурса.
2. POST: используется для создания ресурса.
3. PUT: используется для обновления ресурса.
4. DELETE: используется для удаления ресурса.

Чтобы производить работу с графом в определенном виде, создадим сущности графа: GraphMatrix и TEdge. Так как эти классы будут необходимы для работы, как клиентской части, так и серверной выделим их в отдельный проект. Назовем его Entities. GraphMatrix будет описывать весь граф в целом:

```
public class GraphMatrix
{
    public int[] pointerB; // указатели на начало списка связанных ребер
    public int[] column; // индексы связанных вершин
    public int[] value; // веса ребер
    public int sizeV; // количество вершин
    public int sizeE; // количество ребер
}
```

А TEdge является шаблоном ребра, в котором описывается конструктор и компаратор для сортировки ребер – он нам потребуется в дальнейшем.

```
public class TEdge : IComparable<TEdge>
{
    public int row;
    public int col;
    public int val;
    public TEdge(int r, int c, int v)
    {
        this.row = r;
        this.col = c;
        this.val = v;
    }

    public int CompareTo(TEdge obj)
    {
        if (this.row > obj.row)
            return 1;
        else if
            (this.row < obj.row)
            return -1;
        else return 0;

    }
```

Добавление модели.

Теперь добавим в Обозревателе Решений в папку Models необходимый класс, описывающий сущность - класс Collection. В нем мы произведем парсинг файла, в котором описывается граф

```
string[] q = System.IO.File.ReadAllLines (ConfigurationManager.AppSettings ["filePath"]);
        foreach (string line in q)
        {
            var myArray = line.Split(' ');
            edges.Add(new TEdge(int.Parse(myArray[0])-1, int.Parse(myArray[1])-1,
int.Parse(myArray[2])));
        }
```

и приведение графа к виду, соответствующему сущности GraphMatrix.

```
        foreach(var item in edges)
        {
            int row = item.row;
            graph.pointerB[row]++;
        }
        int sum = 0;
        for (int i = 0; i < graph.sizeV; i++)
        {
            int tmp = graph.pointerB[i];
            graph.pointerB[i] = sum;
            sum += tmp;
        }
        graph.pointerB[graph.sizeV] = sum;
        int[] counter = new int[graph.sizeV];
        for (int i = 0; i < graph.sizeV; i++)
            counter[i] = 0;
        foreach (var item in edges)
        {

            int pos;
            int row = item.row;
            int col = item.col;
            int val = item.val;
            pos = graph.pointerB[row] + counter[row];
            graph.column[pos] = col;
            graph.value[pos] = val;
            counter[row]++;
        }
```

Так же удалим кратные ребра, оставим ребра с наименьшим весом.

```
for ( int i = 0; i != edges.Count - 1;)
        {
            int j = i + 1;
    if (((edges[i]).row == (edges[j]).row) && (edges[i].col == edges[j].col))
            {
                if (edges[j].val < edges[i].val)
                    edges.Remove(edges[i]);
                else
                {
```

```
                edges.Remove(edges[j]);
                i--;
            }
        }
        else
            i++;
    }
```

Добавление контроллера.

В WEB API контроллер – это объект, позволяющий посылать HTTP-запросы. Произведем его добавление в папку Controllers. Этот контролер необходим для получения с сервера данных, описывающих граф.

```
public class NotesController : ApiController
  {
    public IHttpActionResult Get()
    {
      return Json(Collection.graph);
    }
  }
```

Класс ApiController, который является основой контроллеров API, узнает из маршрута, какой контроллер должен обрабатывать запрос, и использует метод HTTP для поиска подходящих методов действий. По соглашению методы действий контроллеров API должны содержать приставку с именем метода действия, который они поддерживают, и тип модели, с которым они работают. Но это только соглашение, поскольку Web API найдет любой метод действия, в имени которого указан метод HTTP, использовавшийся в запросе.

HttpResponseMessage: если метод возвращает объект HttpResponseMessage, то Web API преобразует возвращаемое значение в текст ответа HTTP, то есть возвратит данные в определенном формате - JSON.

2.2.1 JavaScript Object Notation

JavaScript Object Notation (JSON) – это предпочтительный формат структурирования данных, чтобы эти данные впоследствии могли быть считаны разными приложениями, в том числе можно создать форматированные данные с помощью JSON в одном языке программирования и прочитать их в другом.

С помощью контроллера производится сериализация данных - преобразование объекта какого-либо типа в унифицированную бинарную или текстовую последовательность для облегчения передачи его через транспортный протокол. При вызове метода GET по URL http://localhost:52566/api/Notes на рисунке 6 видим GraphMatrix, представленный в формате JSON.

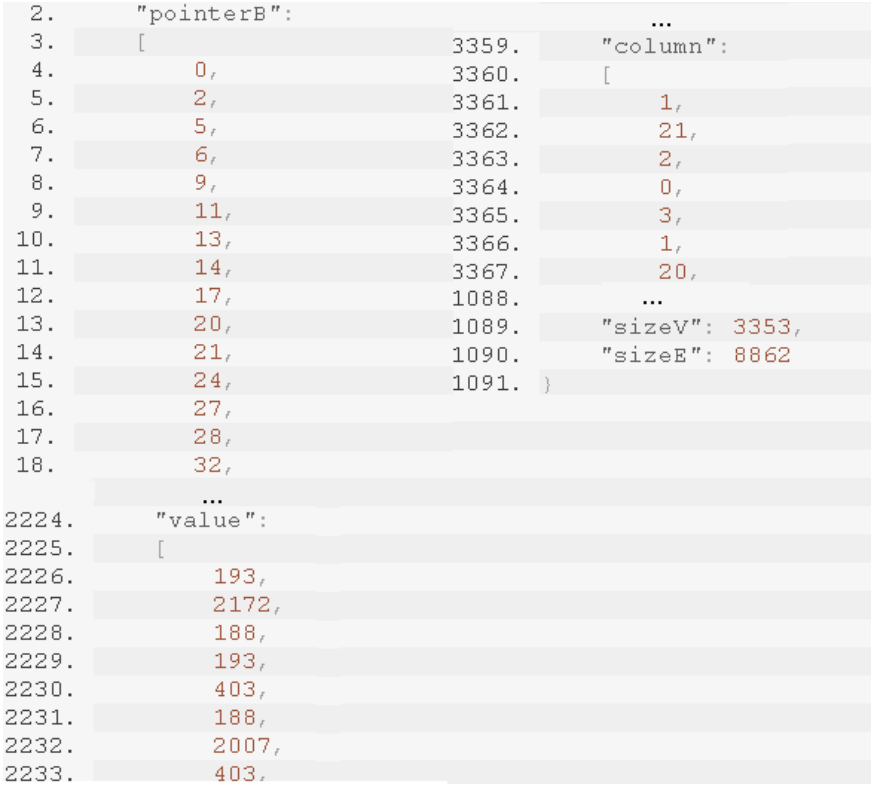

Рисунок 7. GraphMatrix в формате хранения JSON

Для того чтобы получить данные с сервера воспользуемся асинхронным методом.

```
public static async Task DoJob()
    {

        using (var client = new HttpClient())
        {

            var response = await client.GetAsync ("http://localhost:52566/api/Notes" , HttpComple-
tionOption.ResponseHeadersRead)
.ConfigureAwait(false);
          if (response.StatusCode == HttpStatusCode.OK)
          {
            var note = await response.Content.ReadAsAsync<GraphMatrix>();
            int[] up = new int[note.sizeV * note.sizeV];
            int[] dist = new int[note.sizeV * note.sizeV];
```

```
    }
}
    }
```

Асинхронность необходима для действий, которые потенциально являются блокирующими, например, когда приложение получает доступ к веб-ресурсу, который иногда осуществляется медленно или с задержкой. Если такое действие блокируется в пределах синхронного процесса, все приложение вынуждено ожидать. В асинхронном процессе приложение может перейти к следующей операции, не зависящей от веб-ресурса, до завершения блокирующей задачи.

Десериализацию(восстановление ранее сериализованного объекта в первоначальный вид) берёт на себя инфраструктура ASP.NET Web API.

С этими данными клиент уже может непосредственно работать и производить необходимые вычисления.

2.3 Windows Forms

Для просмотра получившихся результатов работы алгоритма Флойда применим рекурсивный метод Way.

```
public static void Way(int[] up, int i, int j, int Size)
    {
        int k = up[i * Size + j];
        if (k != i) // путь существует
        {
            Way(up, i, k, Size); // рекурсивно восстанавливаем путь между вершинами i и k
            vertex.Add(k + 1);
            Way(up, k, j, Size); // и рекурсивно восстанавливаем путь между k и j
        }
         }
```

Отобразим получившиеся результаты работы алгоритма по нахождению пути между конкретными вершинами при помощи графического интерфейса – Windows Forms.

Windows Forms – интерфейс программирования приложений (API), отвечающий за графический интерфейс пользователя и являющийся частью Microsoft .NET Framework. Данный интерфейс упрощает доступ к элементам интерфейса Microsoft Windows за счет создания обёртки для существующего Win32 API в управляемом коде. Причём управляемый код — классы,

реализующие API для Windows Forms, не зависят от языка разработки. То есть программист одинаково может использовать Windows Forms как при написании ПО на C#, C++, так и на VB.Net, J# и др.

Рисунок 8. Путь между вершинами 98 и 2000

Заключение

На конкретном примере графа в виде модели карты дорог города Рим был изучен алгоритм Флойда. Показано, что алгоритм не зависит от размера входных данных и работает даже на графе большой размерности.

В приложении так же применена клиент-серверная архитектура, где серверная часть написана по необходимым условиям REST-архитектуры с реализацией в ASP.NET Web API, а клиентская часть использует интерфейс программирования приложений – Windows Forms.

Результатом работы является вывод пути между двумя конкретными вершинами.

Список использованной литературы

1. Гергель В. П., Баркалов К. А., Мееров И. Б. и др. Параллельные вычисления: технологии и численные методы: учебное пособие. В 4 т. Т. 2. – Н. Новгород: Изд-во Нижегородского госуниверситета, 2013. 367 с.
2. Алгоритм Флойда-Уоршелла // Habrahabr [Электронный ресурс]. – Режим доступа: http://habrahabr.ru/post/105825/. Дата обращения: 21.05.2016.
3. Разреженная матрица // Википедия [Электронный ресурс]. – Режим доступа: http://en.wikipedia.org/wiki/Разреженная_матрица. Дата обращения: 21.05.2016.
4. Набор тестов для проверки алгоритмов поиска кратчайших путей [Электронный ресурс]. – Режим доступа: http://www.dis.uniroma1.it/challenge9/download.shtml. Дата обращения: 21.05.2016.
5. R. T. Fielding. Architectural Styles and the Design of Network-based Software Architectures [Электронный ресурс]. – Режим доступа: http://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm. Дата обращения: 21.05.2016.
6. REST vs SOAP [Электронный ресурс]. – Режим доступа: https://habrahabr.ru/post/158605/. Дата обращения: 21.05.2016.
7. Thomas Erl, Benjamin Carlyle, Cesare Pautasso, Raj Balasubramanian. 5.1 // SOA with REST. — Prentice Hall, 2013.

Приложение А. Реализация алгоритма Флойда

Client.DrawWay.cpp

Program.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using System.Windows.Forms;

namespace Client.DrawWay
{
    static class Program
    {
        /// <summary>
        /// Главная точка входа для приложения.
        /// </summary>
        [STAThread]
        static void Main()
        {
            Application.EnableVisualStyles();
            Application.SetCompatibleTextRenderingDefault(false);
            Application.Run(new Form1());
        }
    }
}
```

Trio.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace Client.DrawWay
{
    public class Trio
    {
        public int vertex;
        public float x;
        public float y;
        public Trio(float x, float y, int vertex)
        {
            this.x = x;
            this.y = y;
            this.vertex = vertex;
        }
    }
}
```

Form1.cs

```
using System;
```

```
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using Entities;
using System.Net;
using System.Net.Http;
using System.Threading.Tasks;
using System.IO;
using System.Windows.Forms;

namespace Client.DrawWay
{

    public partial class Form1 : Form
    {
        public static List<Trio> masTrio = new List<Trio>();
        public static List<int> vertex = new List<int>();
        public Form1()
        {
            InitializeComponent();

            DoJob().Wait();


        }
        public static async Task DoJob()
        {

            using (var client = new HttpClient())
            {

                var response = await client.GetAsync ("http://localhost:52566/api/Notes", HttpComple-
tionOption.ResponseHeadersRead)
.ConfigureAwait(false);
                if (response.StatusCode == HttpStatusCode.OK)
                {
                  var note = await response.Content.ReadAsAsync<GraphMatrix>();
                    int[] up = new int[note.sizeV * note.sizeV];
                    int[] dist = new int[note.sizeV * note.sizeV];
                    FloydWarshall(note, up, dist);
                    Way(up, 100, 2000, 3353);
                }
            }
        }

        public static int Min(int A, int B)
        {
            int Result = (A < B) ? A : B;
            if ((A < 0) && (B >= 0)) Result = B;
            if ((B < 0) && (A >= 0)) Result = A;
            if ((A < 0) && (B < 0)) Result = -1;
            return Result;
        }
```

```
public static void FloydWarshall(GraphMatrix gr, int[] up, int[] dist)
{
    int j;
    // переменные прохода по окрестности вершины графа
    int okr_s, okr_f, okr_i;

    int n = gr.sizeV;

    for (int i = 0; i < n * n; i++)
    {
        dist[i] = int.MaxValue;
    }

    for (int i = 0; i < n; i++)
    {
        dist[i * n + i] = 0;
        up[i * n + i] = i;
    }

    for (int i = 0; i < n; i++)
    {
        okr_s = gr.pointerB[i];
        okr_f = gr.pointerB[i + 1];
        for (okr_i = okr_s; okr_i < okr_f; okr_i++)
        {
            j = gr.column[okr_i];
            if (dist[i * n + j] > gr.value[okr_i])
                dist[i * n + j] = gr.value[okr_i];
            up[i * n + j] = i;
        }
    }

    for (int k = 0; k < n; k++)
        for (int i = 0; i < n; i++)
            for (j = 0; j < n; j++)
                if (dist[i * n + j] - dist[k * n + j] > dist[i * n + k])
                {
                    dist[i * n + j] = dist[i * n + k] + dist[k * n + j];
                    up[i * n + j] = up[k * n + j];
                }
}
public static void Way(int[] up, int i, int j, int Size)
{

    int k = up[i * Size + j];
    if (k != i)
    {

        Way(up, i, k, Size);
        vertex.Add(k + 1);
        Way(up, k, j, Size);
    }
}
```

```
        public static void returnAllCoordinates()
        {

            int i = 0;
            float x = 30;
            float y = 30;
            foreach (var item in vertex)
            {
                i++;
                if (i == 1)
                {
                    masTrio.Add(new Trio(x, y, item));
                }
                else if (i < 16)
                {
                    x += 40;
                    masTrio.Add(new Trio(x, y, item));
                }
                else if (i == 16)
                {
                    y += 60;
                    masTrio.Add(new Trio(x, y, item));
                }
                else if (i < 31)
                {
                    x -= 40;
                    masTrio.Add(new Trio(x, y, item));
                }
                else if (i == 31)
                {
                    i = 1;
                    y += 60;
                    masTrio.Add(new Trio(x, y, item));
                }
            }

        }


        private void panel1_Paint_2(object sender, PaintEventArgs e)
        {
            Pen blackPen = new Pen(Brushes.Black, 10);
            returnAllCoordinates();
            for (int i = 0; i < masTrio.Count - 1; i++)
            {
                PointF point = new PointF(Convert.ToInt32(masTrio[i].x + 5), Con-
vert.ToInt32(masTrio[i].y - 20));
                e.Graphics.DrawLine(blackPen, masTrio[i].x, masTrio[i].y, masTrio[i].x + 1, masTrio[i].y);
                e.Graphics.DrawLine(Pens.Black, masTrio[i].x, masTrio[i].y, masTrio[i + 1].x, masTrio[i +
1].y);
                e.Graphics.DrawString(masTrio[i].vertex.ToString(), new Font("Aerial", 10), Brush-
es.Black, point);

            }
            PointF pointLast = new PointF(Convert.ToInt32(masTrio[masTrio.Count - 1].x + 5), Con-
vert.ToInt32(masTrio[masTrio.Count - 1].y - 20));
```

```
            e.Graphics.DrawString(masTrio[masTrio.Count - 1].vertex.ToString(), new Font("Aerial",
10), Brushes.Black, pointLast);
            e.Graphics.DrawLine(blackPen, masTrio[masTrio.Count - 1].x, masTrio[masTrio.Count -
1].y, masTrio[masTrio.Count - 1].x + 1, masTrio[masTrio.Count - 1].y);
        }

    }
}
```

Entities.cpp

Graph.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace Entities
{
    public class GraphMatrix
    {
        public int[] pointerB; // указатели на начало списка связанных ребер
        public int[] column; // индексы связанных вершин
        public int[] value; // веса ребер
        public int sizeV; // количество вершин
        public int sizeE; // количество ребер
    }
}
```

TEdge.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;

namespace Entities
{
    public class TEdge : IComparable<TEdge>
    {
        public int row;
        public int col;
        public int val;
        public TEdge(int r, int c, int v)
        {
            this.row = r;
            this.col = c;
            this.val = v;
        }

        public int CompareTo(TEdge obj)
        {
            if (this.row > obj.row)
                return 1;
            else if
                (this.row < obj.row)
                return -1;
            else return 0;

        }
```

```
    }
}
```

Server.cpp

Controllers.cs

```
using Entities;
using Models;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Net;
using System.Net.Http;
using System.Web.Http;

namespace Controllers
{
    public class NotesController : ApiController
    {
        public IHttpActionResult Get()
        {
            return Json(Collection.graph);
        }

    }
}
```

Collection.cs

```
using Entities;
using System;
using System.Collections.Generic;
using System.Configuration;
using System.Linq;

namespace Models
{
    public class Collection
    {

        public static List<TEdge> edges = new List<TEdge>();

        public static GraphMatrix graph { get; set; } = new GraphMatrix();

        static Collection()
        {
            string[] q = System.IO.File.ReadAllLines(ConfigurationManager.AppSettings["filePath"]);
            foreach (string line in q)
            {
                var myArray = line.Split(' ');
                edges.Add(new TEdge(int.Parse(myArray[0])-1, int.Parse(myArray[1])-1,
int.Parse(myArray[2])));
            }
            edges.Sort();
```

```
            for ( int i = 0; i != edges.Count - 1;)
            {
                int j = i + 1;
                if (((edges[i]).row == (edges[j]).row) && (edges[i].col == edges[j].col))
                {
                    if (edges[j].val < edges[i].val)
                        edges.Remove(edges[i]);
                    else
                    {
                        edges.Remove(edges[j]);
                        i--;
                    }
                }
                else
                    i++;
            }
            int sizeV = 3353;
            graph.sizeV = sizeV;
            graph.sizeE = edges.Count;
            graph.pointerB = new int[graph.sizeV + 1];
            graph.column = new int[graph.sizeE];
            graph.value = new int[graph.sizeE];
            foreach(var item in edges)
            {
                int row = item.row;
                graph.pointerB[row]++;
            }
            int sum = 0;
            for (int i = 0; i < graph.sizeV; i++)
            {
                int tmp = graph.pointerB[i];
                graph.pointerB[i] = sum;
                sum += tmp;
            }
            graph.pointerB[graph.sizeV] = sum;
            int[] counter = new int[graph.sizeV];
            for (int i = 0; i < graph.sizeV; i++)
                counter[i] = 0;
            foreach (var item in edges)
            {

                int pos;
                int row = item.row;
                int col = item.col;
                int val = item.val;
                pos = graph.pointerB[row] + counter[row];
                graph.column[pos] = col;
                graph.value[pos] = val;
                counter[row]++;
            }
        }

    }
}
```