ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ
ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
«САНКТ-ПЕТЕРБУРГСКИЙ ПОЛИТЕХНИЧЕСКИЙ
УНИВЕРСИТЕТ ПЕТРА ВЕЛИКОГО»
Институт машиностроения, материалов и транспорта
Высшая школа автоматизации и робототехники

КУРСОВАЯ РАБОТА

Способы представления графов
по дисциплине «Объектно-ориентированное программирование»

Выполнил: студент гр. 3331506/90401 Колесов С.А.
Преподаватель: Ананьевский М.С.

Санкт-Петербург
2022

Оглавление

Введение

Граф - это топологическая модель, которая состоит из множества вершин и множества соединяющих их рёбер. При этом значение имеет только сам факт, какая вершина с какой соединена. В свою очередь вершина - точка в графе, отдельный объект, а ребро - неупорядоченная пара двух вершин, которые связаны друг с другом.

Графы являются очень полезной в программировании структурой, поскольку зачастую задачи компьютерной науки можно представить в виде графа и решить с помощью одной из его техник. Графы находят широкое применение в современной науке и технике. Они используются и в естественных науках (физике и химии) и в социальных науках (например, социологии), но наибольших масштабов применение графов получило в информатике и сетевых технологиях.

Для удобной работы с графами, их представляют в памяти различными способами, рассмотрим основные способы представления графов и оценим их характеристики по памяти и скорости доступа к элементам.

Особенности графов

Простой граф $\boldsymbol{G}(\boldsymbol{V},\ \boldsymbol{E})$ – совокупность двух множеств – непустого множества V и множества E неупорядоченных пар различных элементов множества V. Множество V называется множеством вершин, а множество E называется множеством ребер. Пример визуального представления простого графа показан на рисунке 1.

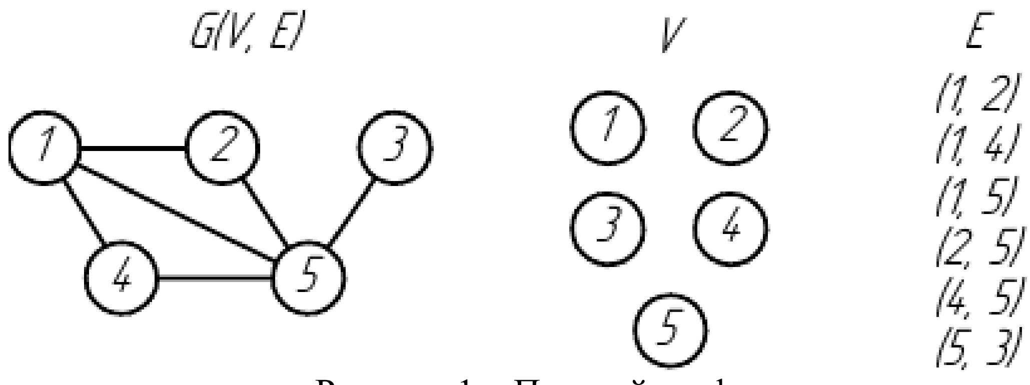


Рисунок 1 – Простой граф

Если в графе $\boldsymbol{G(V,\ E)}$ в множестве ребер *E* разрешены элементы типа $\boldsymbol{e} = \{\boldsymbol{v},\ \boldsymbol{v}\}$, то такой граф называется псевдографом. Другими словами если в графе ребра могут быть петлями, то есть начинаться и заканчиваться в одной вершине, то такой граф $\boldsymbol{G(V,\ E)}$ называется псевдографом. Пример псевдографа представлен на рисунке 2.

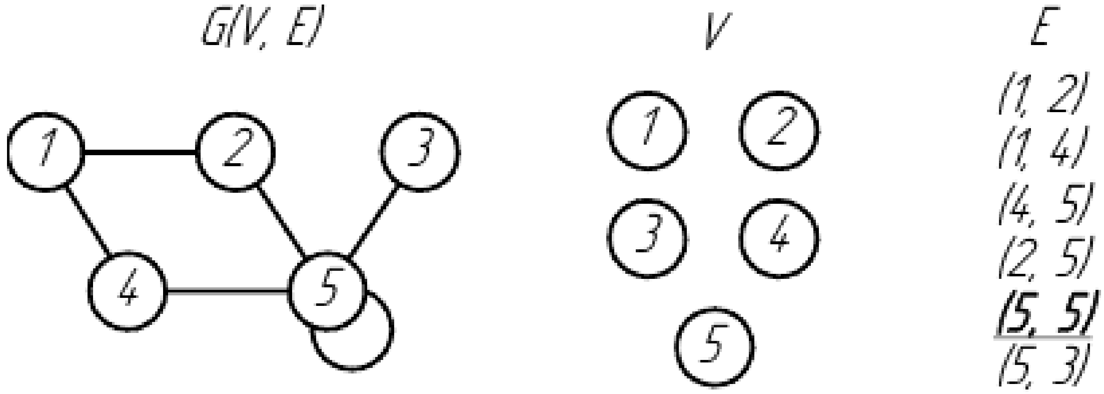


Рисунок 2 – Псевдограф

Если в графе $\boldsymbol{G(V,\ E)}$ в множество ребер *E*, содержит одинаковые элементы, то есть если *E* не множество, а семейство, то такие элементы называются кратными ребрами, а граф $\boldsymbol{G(V,\ E)}$ называется мультиграфом. Пример мультиграфа представлен на рисунке 3.

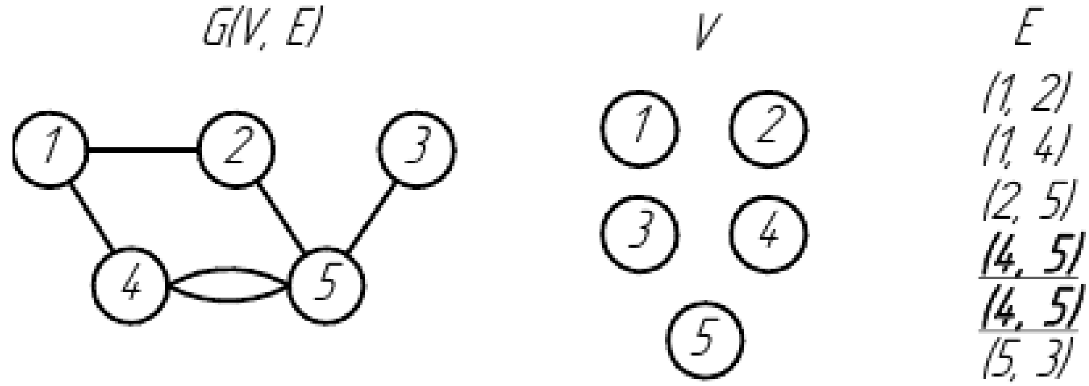


Рисунок 3 – Мультиграф

Если множество ребер *E* в графе $\boldsymbol{G(V,\ E)}$ – множество упорядоченых пар различных элементов множества *V*, то такой граф называется ориентированным или же орграфом. Тогда множество *E* будет состоять из элементов $\boldsymbol{e} = \{\boldsymbol{v_1},\ \boldsymbol{v_2}\}$, где вершина $\boldsymbol{v_1}$ – начало дуги, а $\boldsymbol{v_2}$ – конец дуги. Пример ориентированного графа представлен на рисунке 4.

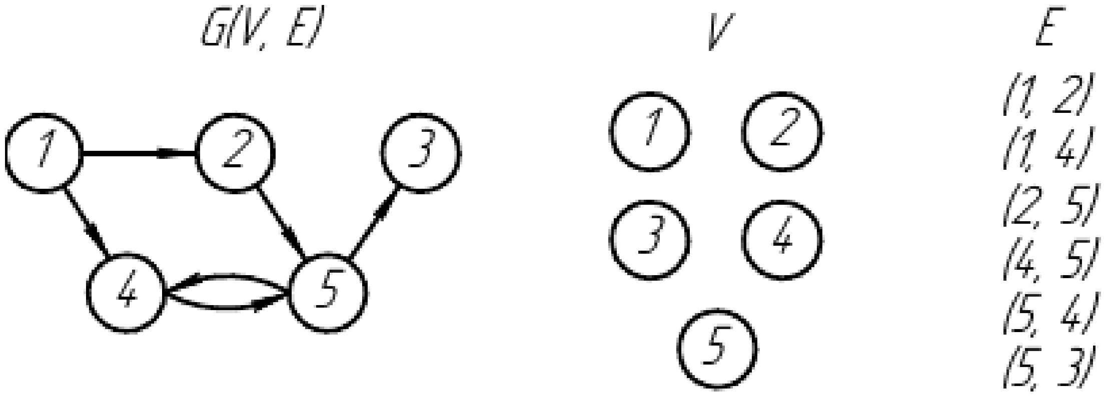


Рисунок 4 – Ориентированный граф

Если в графе присутствуют как ориентированные так и не ориентированные ребра, то есть граф является совокупностью трех множеств $\boldsymbol{G}(\boldsymbol{V},\ \boldsymbol{E},\ \boldsymbol{U})$: множества вершин *V*, множества упорядоченных пар различных вершин *E* и множества неупорядоченных пар различных вершин *U*. Пример смешанного графа представлен на рисунке 5.

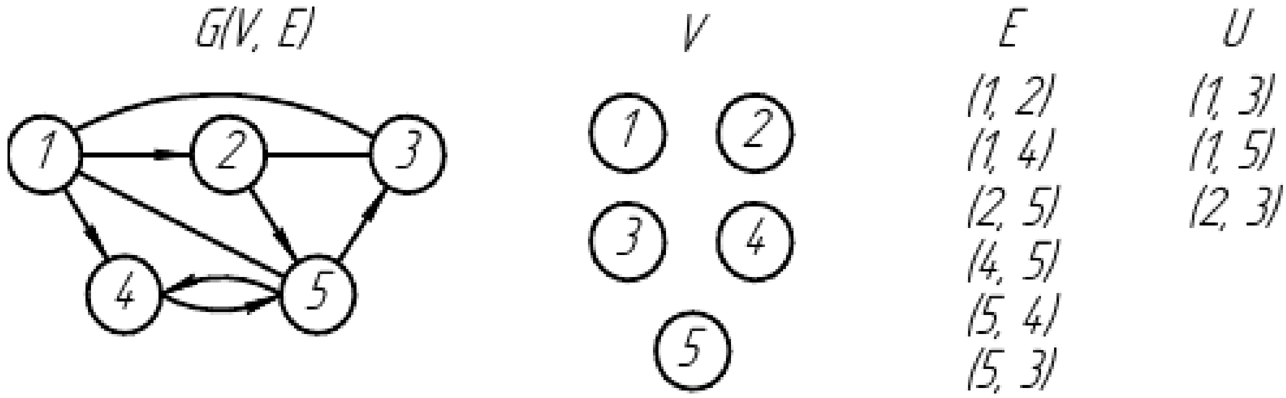


Рисунок 5 – Смешанный граф

Если каждому ребру графа поставлено в соответствие некоторое число – вес ребра, тогда граф называется взвешенным. В таком случае множество *E* будет состоять из элементов $\boldsymbol{e} = \{\boldsymbol{v_1},\ \boldsymbol{v_2},\ \boldsymbol{w}\}$, где $\boldsymbol{w}$ – вес ребра. Пример взвешенного графа представлен на рисунке 6.

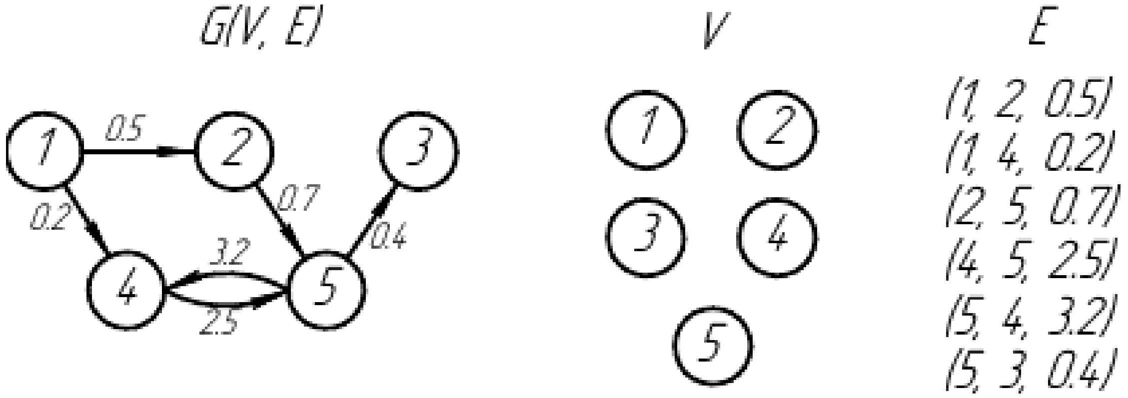


Рисунок 6 – Взвешенный граф

Если каждым ребром графа $\boldsymbol{G(V,\ E)}$ могут соединяться не только две вершины, но любое подмножество вершин графа, то такой граф называется гиперграфом. В таком случае E - семейство непустых (необязательно различных) подмножеств множества V, называемых рёбрами гиперграфа. Гиперграф относится как к теории множеств так и к теории графов, поэтому его стоит учитывать как отдельный вид графа, так как с его помощью решаются определенные типы задач. Пример гиперграфа представлен на рисунке 7.

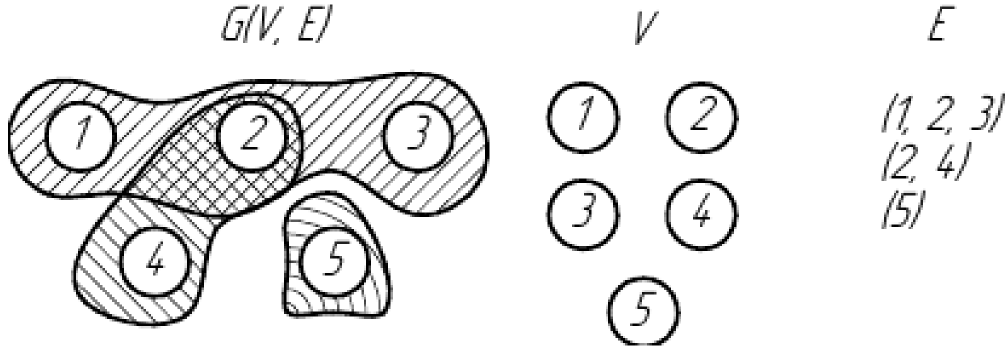


Рисунок 7 - Гиперграф

Способы представления графов

В зависимости от вида графа, его сложности и характера работы с данными о графе используют различные способы представления графа. Основными способами представления графа являются:

- Матрица смежности;
- Матрица инцидентности;
- Список смежности;
- Список рёбер(инцидентности);

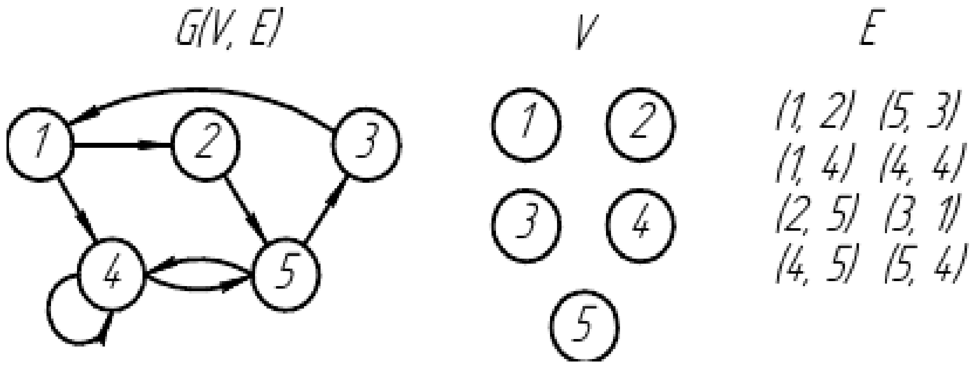
Рассмотрим каждый способ подробнее, и определим какие преимущества имеют те или иные способы представления графа.

Матрица смежности

Смежность – понятие, используемое только в отношении двух ребер или в отношении двух вершин: два ребра инцидентные одной вершине, называются смежными; две вершины, инцидентные одному ребру, также называются смежными.

Граф $\boldsymbol{G}(\boldsymbol{V},\ \boldsymbol{E})$ представляется в виде матрицы A размером $\boldsymbol{n} \times \boldsymbol{n}$, где n – количество вершин в которой столбцы и строки соответствуют вершинам графа. В случае простого графа в элементе матрицы $\boldsymbol{a_{i,j}}$ стоит 0 если между i-ой и j-ой вершиной нет ребра и 1 если оно есть, то есть вершины смежны между собой. Таким образом матрица смежности для простого графа – бинарная матрица, которая содержит нули на главной диагонали. Если граф неориентированный, то матрица смежности симметрична относительно главной диагонали, если ориентированный – нет, в первом случае объем используемой памяти можно сократить вдвое $\boldsymbol{\theta}(|\boldsymbol{V}|^2/2)$. В случае псевдо и мультиграфов элемент матрицы $\boldsymbol{a_{i,j}}$ равен числу ребер из i-ой вершины в j-ую, в свою очередь петли соответствуют диагональным элементам матрицы и в случае ориентированного графа считаются за два ребра. Если граф взвешенный элементы матрицы смежности вместо чисел 0 и 1, указывающих на присутствие или отсутствие ребра, содержат веса самих ребер.

Матрица смежности предпочтительна только в случае плотных графов с большим числом ребер, так как она требует объем памяти $\boldsymbol{\theta}(|\boldsymbol{V}|^2)$, что критично для больших графов. Если граф разрежён, то большая часть памяти напрасно будет тратиться на хранение нулей, зато в случае неразреженных графов матрица смежности достаточно компактно представляет граф в памяти, используя примерно $\boldsymbol{n^2}$ бит памяти, по одному биту на элемент, что может быть на порядок лучше других способов представления. Однако уделять по одному биту на элемент возможно только при невзвешенном графе и если не присутствуют кратные ребра, тогда матрица бинарна. Но в противном случае на один элемент может приходиться несколько байт и тогда матрица смежности очень неэффективный по памяти способ представления. Также если граф взвешенный и в нем имеются кратные ребра, то такой граф нельзя представить в виде матрицы смежности. Пример представления графа в виде матрицы смежности показан на рисунке 8.



Матрица смежности

	1	2	3	4	5
1	0	1	0	1	0
2	0	0	0	0	1
3	1	0	0	0	0
4	0	0	0	1	1
5	0	0	1	1	0

Рисунок 8 – Матрица смежности

Достоинства матрицы смежности:

- Сложность проверки наличия ребра между двумя вершинами $\boldsymbol{O(1)}$;
- Эффективное использование памяти для плотных графов.

Недостатки матрицы смежности:

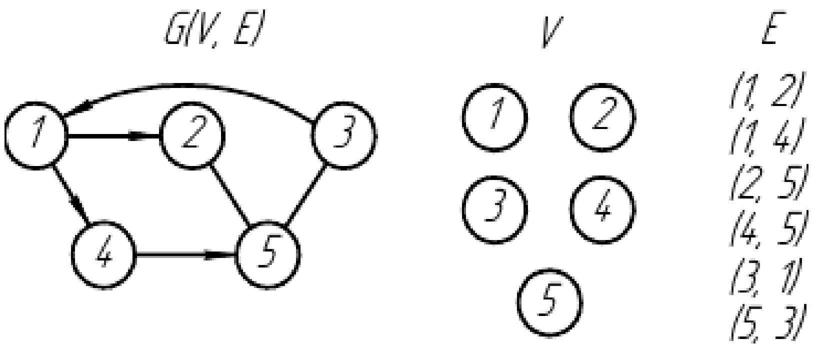
- Объем памяти $\boldsymbol{\theta}(|\boldsymbol{V}|^{\boldsymbol{2}})$, что критично для больших неплотных графов;
- Сложность перебора всех вершин смежных с данной: $\boldsymbol{O}(|\boldsymbol{V}|)$;
- Нельзя представить взвешенный мультиграф;
- Нельзя представить гиперграф;

Матрица инцидентности

Инцидентность – понятие, используемое только в отношении ребра и вершины: две вершины (или два ребра) инцидентными быть не могут. Если $\boldsymbol{v_1}$, $\boldsymbol{v_2}$ – вершины, а $\boldsymbol{e} = \{\boldsymbol{v_1},\ \boldsymbol{v_2}\}$ – соединяющее их ребро, тогда вершина $\boldsymbol{v_1}$ и ребро $\boldsymbol{e}$ инцидентны, также как $\boldsymbol{v_2}$ и $\boldsymbol{e}$.

Граф $\boldsymbol{G}(\boldsymbol{V},\ \boldsymbol{E})$ представляется в виде матрицы *A* размером $\boldsymbol{n} \times \boldsymbol{m}$, где *n* – количество вершин, а *m* – количество вершин в которой столбцы матрицы соответствуют ребрам, строки - вершинам графа. В случае простого графа в элементе матрицы $\boldsymbol{a_{i,j}}$ стоит 0 если между i-ой вершиной и j-ым ребром нет инцидентности и наоборот 1 если она есть. В случае ориентированного графа каждой дуге $\boldsymbol{e} = \{\boldsymbol{v_1},\ \boldsymbol{v_2}\}$, ставится в соответствующем *e* столбце: 1 в строке вершины $\boldsymbol{v_1}$, и -1 в строке вершины $\boldsymbol{v_2}$, если связей между ребром и вершиной нет, то в соответствующую ячейку ставится 0. В случае если представляется псевдограф, то есть присутствуют петли, то в столбец ставится одна 1, иначе в матрице в каждом столбце должно быть ровно две ненулевых ячейки. Если представляется гиперграф, то в столбце тоже может быть отличное от двух число ненулевых ячеек. В случае взвешенного графа вместо 1 и -1 в соответствующие ячейки записывается вес самих ребер.

Матрица инцидентности может использоваться для любых видов графов. Так как в столбце данной матрицы в случае простого графа находится только два ненулевых элемента, то уже при небольшом числе вершин матрица будет разреженной от чего память будет неэффективно использоваться. Однако для невзвешенного и неориентированного графа матрица будет бинарной, от чего ее можно более компактно хранить. Матрица инцидентности не позволяет проверять наличие ребер между двумя вершинами быстрее чем за $O(|E|)$, что довольно долго, а удалять ребро быстрее чем за $O(|V|)$, так как последний столбец копируется в столбец удаляемого ребра. Пример представления графа в виде матрицы инцидентности показан на рисунке 9.



Матрица инцидентности

	1	2	3	4	5	6
1	1	1	0	0	-1	0
2	-1	0	1	0	0	0
3	0	0	0	0	1	1
4	0	-1	0	1	0	0
5	0	0	1	-1	0	1

Рисунок 9 – Матрица инцидентности

Достоинства матрицы смежности:

- Можно представить любой вид графа;
- Удобно определять число входов и выходов в вершину;

Недостатки матрицы смежности:

- Объем памяти $\boldsymbol{\theta}(|\boldsymbol{V}|\cdot|\boldsymbol{E}|)$, матрица разрежена;
- Сложность перебора всех вершин смежных с данной: $\boldsymbol{O}(|\boldsymbol{E}|)$;
- Сложность удаления ребра $\boldsymbol{O}(|\boldsymbol{V}|)$;

Список смежности

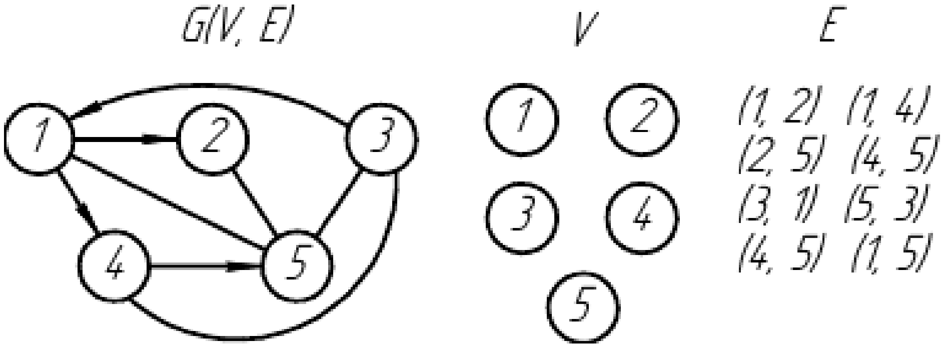
Граф $\boldsymbol{G}(\boldsymbol{V},\ \boldsymbol{E})$ представляется в виде списка где каждой вершине соответствует строка в которой хранится список смежных вершин. Так как количество смежных вершин у различных вершин различно, данная структура представляет собой не матрицу, а список списков. В случае взвешенного графа в списке смежных вершин также указывается вес ребра для каждой смежной вершины. Если граф неориентированный, то из-за того что рассматривается смежность вершин будет иметь место дублирование и сумма длин всех списков будет $2|E|$, при ориентированном графе дублирования не будет и память будет использоваться более эффективно. Недостатком списка смежности является, то что в плотном графе для определения смежности двух вершин, требуется поиск по списку сложностью $\boldsymbol{O}(|\boldsymbol{V}|)$.

Для реализации данной структуры используется несколько способов, различающиеся особенностями ассоциации вершин и коллекциями соседей, и способами представления ребер и вершин:

- Использование хэш-таблицы для ассоциации каждой вершины со списком смежных вершин. Нет явного представления ребер.
- Вершины представляются в массиве, индекс элемента соответствует номеру вершины, каждый элемент массива ссылается на однонаправленный связный список соседних вершин.
- Объектно-ориентированный список смежности, который содержит классы вершин и ребер. В списке вершин, каждый объект вершины содержит ссылку на коллекцию рёбер, а каждый объект ребра содержит ссылки на вершины начала и конца ребра.

В отличии от матричных представлений списки смежности нельзя представить в бинарном виде, поэтому их выгодно использовать, если граф разрежен, так как объем используемой памяти: $\boldsymbol{\theta}(|\boldsymbol{V}|+|\boldsymbol{E}|)$, в отличии от матричных представлений в памяти не хранится преобладающее количество нулей. Также список смежности выгоден в операциях над графом по определению степени вершины, вставке и удалении вершины и обходе графа.

Пример списка смежности представлен на рисунке 10.



Список смежности

1	2	4	5
2	5		
3	1	4	5
4	3	5	
5	2	3	4

Рисунок 10 – Список смежности

Достоинства списка смежности:

- Объем памяти $\boldsymbol{\theta}(|\boldsymbol{V}| + |\boldsymbol{E}|)$, оптимально для не насыщенных графов
- Позволяет быстро перебирать смежные вершины, и обходить граф;

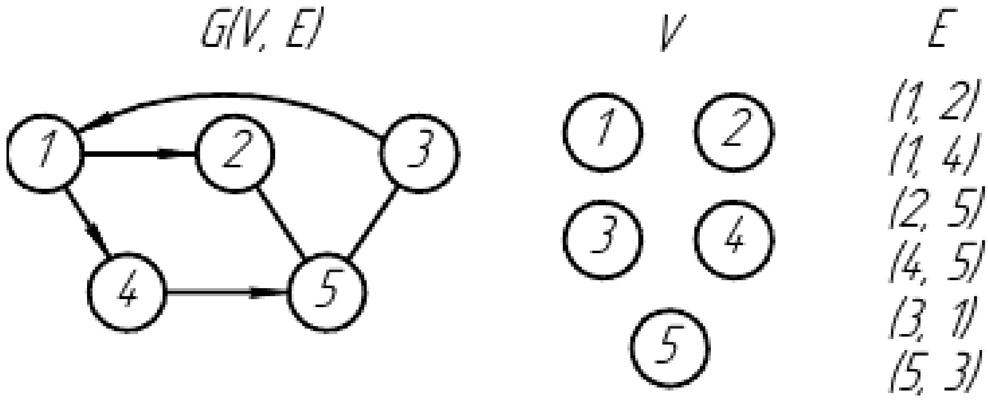
Недостатки списка смежности:

- Не оптимален для насыщенных графов (количество ребер $\sim\boldsymbol{n}^{\boldsymbol{2}}$);
- Для взвешенных графов приходится усложнять структуру списка смежных вершин добавлением веса;
- Сложность определения ребра между вершинами $\boldsymbol{O}(|\boldsymbol{V}|)$.

Список ребер(инцидентности)

Граф $\boldsymbol{G}(\boldsymbol{V},\ \boldsymbol{E})$ представляется в виде списка где каждому ребру соответствует строка в которой хранится список инцидентных данному ребру вершин. В отличии от списка смежности в реализации списка ребер в массиве хранятся не объекты вершин с ссылками на списки смежных вершин, а объекты ребер с ссылками на список инцидентных вершин. Также как и матрица инцидентности, с помощью списка ребер можно представить любой граф, в том числе и гиперграф. В случае взвешенного графа, помимо инцидентных верщин хранится вес ребра, а если граф смешанный то неориентированные ребра записываются в список ребер дважды. В общем случае список ребер представляет из себя таблицу с количеством строк равным количеству ребер и двумя столбцами в случае невзвешенного графа и тремя столбцами в случае взвешенного графа.

Список рёбер наиболее компактный способ представления графов, поэтому его применяют для внешнего хранения или обмена данными. На этом достоинства списка ребер заканчиваются, так как такие операции как: определение ребра между двумя вершинами, удаление вершины, обход графа, определение степени вершины и т.п. делать не оптимально в данном представлении. Пример списка ребер представлен на рисунке 11.



Список ребер

1	1	2
2	1	4
3	4	5
4	3	1
5	2	5
6	5	2
7	3	5
8	5	3

Рисунок 11 – Список рёбер

Достоинства списка смежности:

- Объем памяти $\boldsymbol{\theta}(|\boldsymbol{E}|)$, наиболее компактно;
- Можно представить любой вид графа;

Недостатки списка смежности:

- Не оптимален для насыщенных графов (количество ребер $\sim\boldsymbol{n}^{\boldsymbol{2}}$);
- Операции над графом, за исключением удаления ребра, имеют сложность $\boldsymbol{O}(|\boldsymbol{E}|)$, что критично для насыщенных графов;

Сравнение способов представления

Каждый способ представления графа имеет свои преимущества и недостатки, сравним их по объему памяти и сложности выполнения операций над графом, таких как: обход графа, проверка наличия ребра $\{\boldsymbol{v_1},\ \boldsymbol{v_2}\}$, определение степени вершины, вставка или удаление вершины, вставка или удаление ребра. Результаты сравнения представлены в таблице 1.

Таблица 1 – Сравнение способов представления графа

Операция	Матрица смежности	Матрица инцидентности	Список смежности	Список рёбер
Объем используемой памяти	$\boldsymbol{\theta}(\lvert\boldsymbol{V}\rvert^{2})$	$\boldsymbol{\theta}(\lvert\boldsymbol{V}\rvert\cdot\lvert\boldsymbol{E}\rvert)$	$\boldsymbol{\theta}(\lvert\boldsymbol{V}\rvert+\lvert\boldsymbol{E}\rvert)$	$\boldsymbol{\theta}(\lvert\boldsymbol{E}\rvert)$
Проверка на наличие ребра $\{\boldsymbol{v_1},\ \boldsymbol{v_2}\}$	$\boldsymbol{O}(\boldsymbol{1})$	$\boldsymbol{O}(\lvert\boldsymbol{E}\rvert)$	$\boldsymbol{O}(\lvert\boldsymbol{V}\rvert)$	$\boldsymbol{O}(\lvert\boldsymbol{E}\rvert)$
Определение степени вершины	$\boldsymbol{O}(\lvert\boldsymbol{V}\rvert)$	$\boldsymbol{O}(\lvert\boldsymbol{E}\rvert)$	$\boldsymbol{O}(\boldsymbol{1})$	$\boldsymbol{O}(\lvert\boldsymbol{E}\rvert)$
Вставка или удаление вершины	$\boldsymbol{O}(\lvert\boldsymbol{V}\rvert)$	$\boldsymbol{O}(\lvert\boldsymbol{E}^{2}\rvert)$	$\boldsymbol{O}(\boldsymbol{d})$	$\boldsymbol{O}(\lvert\boldsymbol{E}^{2}\rvert)$
Вставка или удаление ребра	$\boldsymbol{O}(\boldsymbol{1})$	$\boldsymbol{O}(\boldsymbol{1})$	$\boldsymbol{O}(\boldsymbol{1})$	$\boldsymbol{O}(\lvert\boldsymbol{1}\rvert)$
Обход графа	$\boldsymbol{O}(\lvert\boldsymbol{V}\rvert^{2})$	$\boldsymbol{O}(\lvert\boldsymbol{V}\rvert\cdot\lvert\boldsymbol{E}\rvert)$	$\boldsymbol{O}(\lvert\boldsymbol{V}\rvert+\lvert\boldsymbol{E}\rvert)$	$\boldsymbol{O}(\lvert\boldsymbol{E}^{2}\rvert)$
Представление любого вида графа	Нет	Да	Нет	Да

Для представления графа в алгоритмах в основном используется список смежности, благодаря преимуществам в скорости выполнения основных операций, однако для передачи данных о графе и хранении используется список ребер благодаря большей компактности.

Для подтверждения аналитических характеристик способов представления графов была получена зависимость времени выполнения операций удаления вершин и ребер от количества ребер в графе. При получении зависимости использовался граф из 1000 вершин и со случайно составленными ребрами, количество которых менялось от 1 тыс. до 1024 тыс., увеличиваясь на два, поэтому ось количества ребер в графе логарифмическая. Зависимости показывают время затрачиваемое на удаление 10 случайных вершин и на удаление 10 случайных ребер. Графики полученных зависимостей представлены на рисунках 12 и 13.

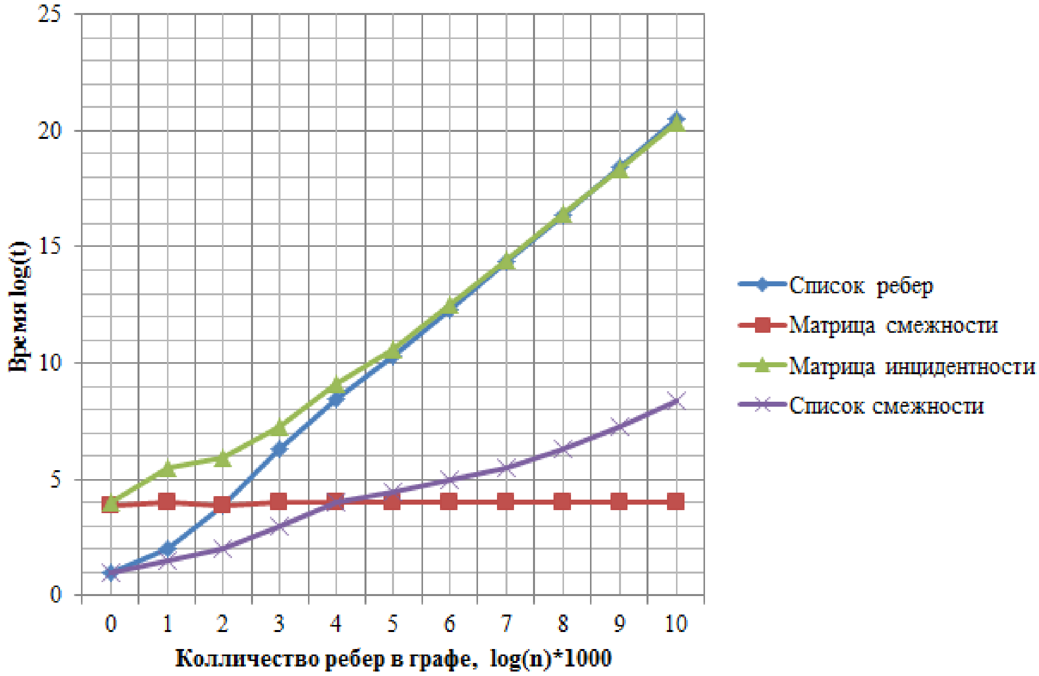


Рисунок 12 – График зависимости времени от количества ребер, для операции удаления вершины, логарифмический масштаб

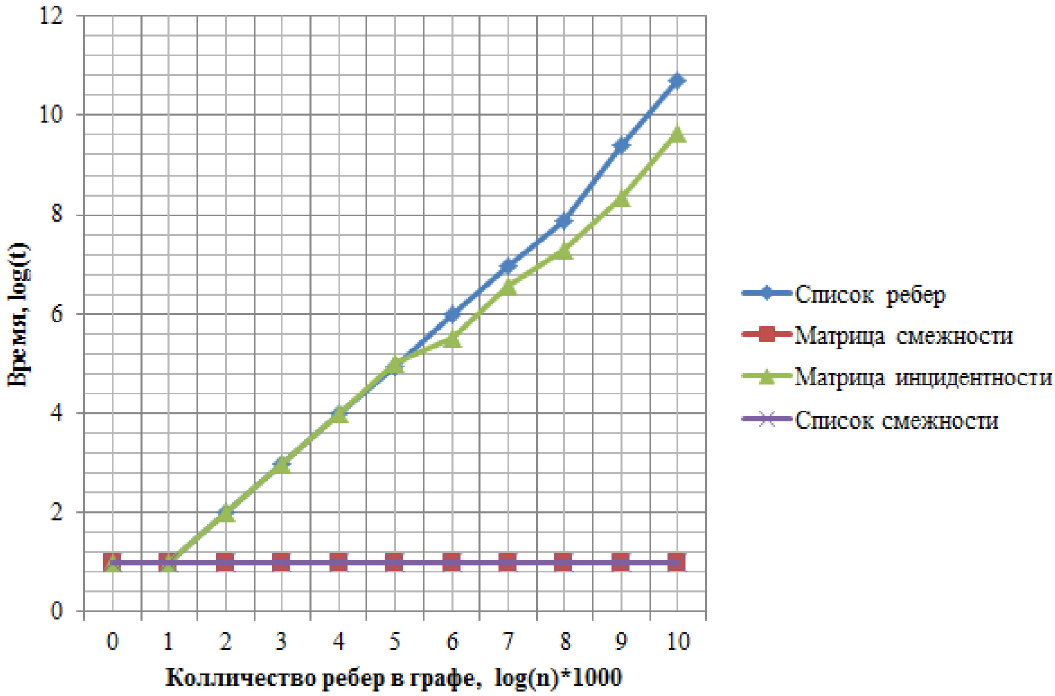


Рисунок 13 – График зависимости времени от количества ребер, для операции удаления ребра, логарифмический масштаб

Можно заключить, что данные графики подтверждают теоретическую сложность выполнения операций удаления ребра и вершины для разных способов представления графа.

Заключение

Для представления графа в алгоритмах в основном используется список смежности, благодаря преимуществам в скорости выполнения основных операций, так как он позволяет быстро получать соседей вершин, однако для передачи данных о графе и хранении используется список ребер благодаря большей компактности. В свою очередь матрицу смежности выгодно использовать для насыщенных графов, но в случае взвешенных графов или мультиграфов, структура усложняется, и объем используемой памяти становится еще больше. Матрица инцидентности используется крайне редко из-за большого объема используемой памяти, однако может применяться для быстрого нахождения циклов в графе.

Список литературы

1. Кормен, Т., Лейзерсон, Ч., Ривест, Р., Штайн, К. Алгоритмы: построение и анализ, под ред. И. В. Красикова. — 2-е изд. — М.: Вильямс, 2005. — 1296 с.
2. Харари Ф. Теория графов. — М.: Мир. — 1973. — 300 с.
3. Граф (математика) [Электронный ресурс] // https://ru.wikipedia.org/wiki/Граф_(математика)#Простой_граф.
4. Графы. Способы представления графа в программе [Электронный ресурс] // https://pro-prof.com/forums/topic/graph-representations.
5. Понятие и представление графа: матрица смежности, список смежности [Электронный ресурс] // https://brestprog.by/topics/graphs.

Приложение 1 – Graph.h

```
#include <iostream>
#include <vector>

#ifndef GRAPH_GRAPH_H
#define GRAPH_GRAPH_H

typedef enum{
    ADJACENCY,
    INCIDENCE,
    ADJACLIST,
    EDGESLIST
} PresentType;

class Graph {
private:

    struct property{
        bool directed = false;
        bool weighted = false;
        bool multiple = false;
  };

    struct property properties;
    PresentType type = EDGESLIST;
    std::vector< std::vector< int > > graph;
    std::vector<int> buf;
    unsigned int num_edges = 0;
    unsigned int num_nodes = 0;

public:
    Graph(Graph& clone);
    Graph(Graph&& clone) noexcept;
    Graph(unsigned int num_nodes, bool weighted = false,
          bool directed = false, bool multiple = false); // Only num_nodes Nodes
without edges
    Graph(unsigned int num_nodes, unsigned int num_edges, bool weighted = false,
          bool directed = false, bool multiple = false); // num_nodes nodes and
read num_edges edges
    Graph& operator= (const Graph&  clone);
    Graph& operator= (const Graph&& clone) noexcept;
    ~Graph();

public:
    void add_edge(unsigned int begin, unsigned int end, unsigned int weight = 1,
bool direct = false);
    void add_node();
    void delete_node(unsigned int n);
    void delete_edge(unsigned int begin, unsigned int end, unsigned int weight =
1, bool direct = false);
    void represent(PresentType type);
    void present_like_EL();
    void present_from_EL(PresentType type);
    void print();
    void free_memory();

};

#endif //GRAPH_GRAPH_H
```

Приложение 2 – Graph.cpp

```
#include "Graph.h"
#include <iostream>

void Graph::free_memory(){
    graph.clear();
    buf.clear();
    num_nodes = 0;
    num_edges = 0;
    type = EDGESLIST;
    properties.weighted = false;
    properties.directed = false;
    properties.multiple = false;
}

Graph::Graph(Graph &clone){
    graph      = clone.graph;
    properties = clone.properties;
    num_edges  = clone.num_edges;
    num_nodes  = clone.num_nodes;
    type       = clone.type;

}

Graph::Graph(Graph &&clone) noexcept{
    graph      = clone.graph;
    properties = clone.properties;
    num_edges  = clone.num_edges;
    num_nodes  = clone.num_nodes;
    type       = clone.type;
    clone.free_memory();
}

Graph& Graph::operator= (const Graph& clone){
    if (this == &clone)
        return *this;

    free_memory();
    graph      = clone.graph;
    properties = clone.properties;
    num_edges  = clone.num_edges;
    num_nodes  = clone.num_nodes;
    type       = clone.type;
    return *this;
}

Graph& Graph::operator= (const Graph&& clone) noexcept{
    graph      = clone.graph;
    properties = clone.properties;
    num_edges  = clone.num_edges;
    num_nodes  = clone.num_nodes;
    type       = clone.type;
    return *this;
}

Graph::Graph(unsigned int num_nodes, bool weighted, bool directed, bool multiple)
{
    this->num_edges = 0;
    this->num_nodes = num_nodes;
    properties.weighted = weighted;
    properties.directed = directed;
    properties.multiple = multiple;
}

Graph::Graph(unsigned int num_nodes, unsigned int num_edges, bool weighted,
                                               bool directed, bool multiple)
```

```
{
    this->num_edges = num_edges;
    this->num_nodes = num_nodes;
    properties.weighted = weighted;
    properties.directed = directed;
    properties.multiple = multiple;

    unsigned int x;
    buf.resize(2);
    if (weighted){
        buf.resize(3);
    }

    std::cout<<"Input "<<num_edges<<" edges"<<std::endl;
    for (int i = 0; i < num_edges; i++)
    {
        if (weighted){
            std::cin>>buf[0]>>buf[1]>>buf[2];
        }
        else{
            std::cin>>buf[0]>>buf[1];
        }
        graph.push_back(buf);
        if (!(directed)){
            x = buf[0];
            buf[0] = buf[1];
            buf[1] = x;
            graph.push_back(buf);
        }
    }
}

Graph::~Graph(){
    this->free_memory();
}

void Graph::add_edge(unsigned int begin, unsigned int end, unsigned int weight,
bool direct)
{
    num_edges++;

    switch (type) {
        case ADJACENCY:
            graph[begin][end] += weight;
            if (!(direct)){
                graph[end][begin] += weight;
            }
            break;

        case INCIDENCE:
            for (int i = 0; i < num_nodes; i++){
                graph[i].push_back(0);
            }
            graph[begin][num_edges-1] = weight;
            if (direct){
                graph[end][num_edges-1] = -weight;
            }
            else{
                graph[end][num_edges-1] = weight;
            }
            break;

        case ADJACLIST:
            if (properties.weighted){
                graph[begin].push_back(end);
                graph[begin].push_back(weight);
                if (!(direct)){
                    graph[end].push_back(begin);
```

```
                    graph[end].push_back(weight);
                }
            }
            else{
                graph[begin].push_back(end);
                if (!(direct)){
                    graph[end].push_back(begin);
                }
            }
            break;

        case EDGESLIST:
            buf.resize(2);
            buf[0] = begin;
            buf[1] = end;
            if (properties.weighted){
                buf.resize(3);
                buf[2] = weight;
            }
            graph.push_back(buf);
            if ((!(direct)&(properties.directed))){
                unsigned int x = buf[0];
                buf[0] = buf[1];
                buf[1] = x;
                graph.push_back(buf);
                num_edges++;
            }
            break;
    }
}

void Graph::add_node()
{
    num_nodes++;
    switch (type) {
        case ADJACENCY:
            buf.resize(num_nodes - 1);
            std::fill(buf.begin(), buf.end(), 0);
            graph.push_back(buf);
            for (int i = 0; i < num_nodes; i++) {
                graph[i].push_back(0);
            }
            break;

        case INCIDENCE:
            for (int i = 0; i < num_edges; i++) {
                graph[i].push_back(0);
            }
            break;

        case ADJACLIST:
            buf.clear();
            graph.push_back(buf);
            break;

        case EDGESLIST:

            break;
    }
}

void Graph::delete_edge(unsigned int begin, unsigned int end, unsigned int weight,
bool direct){
    num_edges--;
    if ((begin >= num_nodes)|(end >= num_nodes)){
        return;
    }
```

```
    switch (type) {
        case ADJACENCY:
            if (graph[begin][end] != 0) {
                graph[begin][end] -= weight;
            }
            if (!(direct)){
                delete_edge(end, begin, weight, true);
            }
            break;

        case INCIDENCE:
            for (int i = 0; i < num_edges; i++) {
                if (graph[i][begin] == weight){
                    if ((direct)&(graph[i][end] == -weight)){
                        graph.erase(graph.begin() + i);
                        break;
                    }
                    if ((!(direct))&(graph[i][end] == weight)){
                        graph.erase(graph.begin() + i);
                        break;
                    }
                }
            }
            break;

        case ADJACLIST:
            if (properties.weighted){
                for (int i = 0; i < graph[begin].size(); i+=2){
                    if ((graph[begin][i] == end)&(graph[begin][i+1] == weight)){
                        graph.erase(graph.begin() + i, graph.begin() + i + 2);
                        break;
                    }
                }
            }
            else{
                for (int i = 0; i < graph[begin].size(); i++){
                    if (graph[begin][i] == end){
                        graph.erase(graph.begin() + i);
                        break;
                    }
                }
            }
            if (!(direct)){
                delete_edge(end, begin, weight, true);
            }
            break;

        case EDGESLIST:
            for (int i = 0; i < graph.size(); i++){
                if ((graph[i][0] == begin)&(graph[i][1] == end)){
                    if (properties.weighted){
                        if (graph[i][2] == weight){
                            graph.erase(graph.begin() + i);
                            break;
                        }
                    }
                    else{
                        graph.erase(graph.begin() + i);
                        break;
                    }
                }
            }
            if ((properties.directed)&((!direct))){
                delete_edge(end, begin, weight, true);
            }
            break;
    }
}
```

```
void Graph::delete_node(unsigned int n)
{
    num_nodes--;
    if (n >= num_nodes){
        return;
    }
    switch (type) {
        case ADJACENCY:
            graph.erase(graph.begin() + n);
            for (int i = 0; i < num_nodes; i++) {
                graph[i].erase(graph[i].begin() + n);
            }
            break;

        case INCIDENCE:
            for (int i = 0; i < num_edges; i++) {
                if (graph[i][n] != 0){
                    graph.erase(graph.begin() + i);
                    num_edges--;
                    i--;
                }
                else{
                    graph[i].erase(graph[i].begin() + n);
                }
            }
            break;

        case ADJACLIST:
            graph.erase(graph.begin() + n);
            for (int i = 0; i < graph.size(); i++) {
                if (properties.weighted){
                    for (int j = 0; j < graph[i].size(); j+=2) {
                        if (graph[i][j] == n){
                            graph[i].erase(graph[i].begin() + j, graph[i].begin()
+ j + 2);

                            j-=2;
                        }
                        if (graph[i][j] > n){
                            graph[i][j]--;
                        }
                    }
                }
                else{
                    for (int j = 0; j < graph[i].size(); j++) {
                        if (graph[i][j] == n){
                            graph[i].erase(graph[i].begin() + j);
                            j--;
                        }
                        if (graph[i][j] > n){
                            graph[i][j]--;
                        }
                    }
                }
            }
            break;

        case EDGESLIST:
            for (int i = 0; i < graph.size(); i++){
                if ((graph[i][0] == n)|(graph[i][1] == n)){
                    graph.erase(graph.begin() + i);
                    i--;
                }
                if (graph[i][0] > n){
                    graph[i][0] -= 1;
                }
                if (graph[i][1] > n){
                    graph[i][1] -= 1;
```

```
                }
            }
            break;
    }
}

void Graph::present_from_EL(PresentType type) {
    unsigned n;
    this->type = type;
    switch (type)
    {
        case ADJACENCY:
            if (properties.weighted & properties.multiple){
                std::cout << "This type of graph can't be present like adjacency
matrix";
                break;
            }
            n = graph.size();
            buf.resize(num_nodes);
            std::fill(buf.begin(), buf.end(), 0);
            graph.resize(n + num_nodes);
            std::fill(graph.begin() + n, graph.end(), buf);
            for (int i = 0; i < n; i++){
                if (properties.weighted){
                    graph[n + graph[i][0]][graph[i][1]] += graph[i][2];
                    if (!(properties.directed)){
                        graph[n + graph[i][1]][graph[i][0]] += graph[i][2];
                    }
                }
                else{
                    graph[n + graph[i][0]][graph[i][1]]++;
                    if (!(properties.directed)){
                        graph[n + graph[i][1]][graph[i][0]]++;
                    }
                }
            }
            graph.erase(graph.begin(), graph.begin() + n);
            break;

        case INCIDENCE:
            n = graph.size();
            buf.resize(num_nodes);
            std::fill(buf.begin(), buf.end(), 0);
            graph.resize(n + num_edges);
            std::fill(graph.begin() + n, graph.end(), buf);
            for (int i = 0; i < n; i++){
                if (properties.weighted) {
                    graph[n + i][graph[i][0]] += graph[i][2];
                    graph[n + i][graph[i][1]] += graph[i][2];
                }
                else {
                    graph[n + i][graph[i][0]]++;
                    graph[n + i][graph[i][1]]++;
                }
                if (properties.directed){
                    graph[n + i][graph[i][1]]*=-1;
                }
            }
            graph.erase(graph.begin(), graph.begin() + n);
            break;

        case ADJACLIST:
            n = graph.size();
            buf.clear();
            graph.resize(n + num_nodes);
            std::fill(graph.begin() + n, graph.end(), buf);
            for (int i = 0; i < n; i++) {
                graph[n + graph[i][0]].push_back(graph[i][1]);
```

```
                if (properties.weighted) {
                    graph[n + graph[i][0]].push_back(graph[i][2]);
                }
                if (!(properties.directed)){
                    graph[n + graph[i][1]].push_back(graph[i][0]);
                    if (properties.weighted) {
                        graph[n + graph[i][1]].push_back(graph[i][2]);
                    }
                }
            }
            graph.erase(graph.begin(), graph.begin() + n);
            break;
        case EDGESLIST:
            break;
    }
}

void Graph::present_like_EL() {
    buf.resize(2);
    if (properties.weighted){
        buf.resize(3);
    }
    switch (this->type)
    {
        case ADJACENCY:
            for (int i = 0; i < num_nodes; i++) {
                for (int j = 0; j < num_nodes; j++) {
                    if (graph[i][j] != 0){
                        buf[0] = i;
                        buf[1] = j;
                        if (properties.weighted){
                            buf[2] = graph[i][j];
                        }
                        if (properties.multiple){
                            for (int k = 0; k < graph[i][j]; k++){
                                 graph.push_back(buf);
                            }
                        }
                        else{
                            graph.push_back(buf);
                        }
                        if (!(properties.directed)){
                            graph[j][i] = 0;
                        }
                    }
                }
            }
            graph.erase(graph.begin(), graph.begin() + num_nodes);
            num_edges = graph.size();
            break;

        case INCIDENCE:
            for (int i = 0; i < num_edges; i++) {
                buf[0] = -1;
                buf[1] = -1;
                for (int j = 0; j < num_nodes; j++) {
                    if (graph[i][j] > 0){
                        if (buf[0] >= 0){
                            buf[1] = j;
                            if (properties.directed){
                                graph.push_back(buf);
                                buf[1] = buf[0];
                                buf[0] = j;
                                graph.push_back(buf);
                                break;
                            }
                        }
                        else{
```

```
                        buf[0] = j;
                    }
                    if (properties.weighted){
                        buf[2] = graph[i][j];
                    }
                }
                if (graph[i][j] < 0){
                    buf[1] = j;
                }
                if ((buf[0]>=0)&(buf[1]>=0)){
                    graph.push_back(buf);
                    break;
                }
            }
        }
        graph.erase(graph.begin(), graph.begin() + num_edges);
        num_edges = graph.size();
        break;

    case ADJACLIST:
        for (int i = 0; i < num_nodes; i++) {
            buf[0] = i;
            if (properties.weighted){
                for (int j = 0; j < graph[i].size(); j+=2) {
                    buf[1] = graph[i][j];
                    buf[2] = graph[i][j+1];
                    graph.push_back(buf);
                    if (!(properties.directed)){
                        for (int k = 0; k < graph[buf[1]].size(); k+=2){
                            if ((graph[buf[1]][k] == i)&(graph[buf[1]][k+1] ==
buf[2])){
                                graph[buf[1]].erase(graph[buf[1]].begin() + k,
graph[buf[1]].begin() + k + 2);
                            }
                        }
                    }

                }
            }
            else{
                for (int j = 0; j < graph[i].size(); j++) {
                    buf[1] = graph[i][j];
                    graph.push_back(buf);
                    if (!(properties.directed)){
                        for (int k = 0; k < graph[buf[1]].size(); k++){
                            if (graph[buf[1]][k] == i){
                                graph[buf[1]].erase(graph[buf[1]].begin() +
k);
                            }
                        }
                    }
                }
            }

        }
        graph.erase(graph.begin(), graph.begin() + num_nodes);
        num_edges = graph.size();
        break;

    case EDGESLIST:
        break;
    }
    this->type = EDGESLIST;
}

void Graph::represent(PresentType type) {
    this->present_like_EL();
    this->present_from_EL(type);
```

```cpp
}

void Graph::print(){
    std::cout << num_nodes << "  " << num_edges << "\n";
    switch (type)
    {
        case ADJACENCY:
            for (int i = 0; i < num_nodes; i++){
                for (int j = 0; j < num_nodes; j++){
                    std::cout << graph[i][j] << "  ";
                }
                std::cout << '\n';
            }
            std::cout << '\n';
            break;
        case INCIDENCE:
            for (int i = 0; i < num_nodes; i++){
                for (int j = 0; j < num_edges; j++){
                    std::cout << graph[j][i] << "  ";
                }
                std::cout << '\n';
            }
            std::cout << '\n';
            break;
        case ADJACLIST:
            for (int i = 0; i < num_nodes; i++){
                for (int j = 0; j < graph[i].size(); j++){
                    std::cout << graph[i][j] << "  ";
                }
                std::cout << '\n';
            }
            std::cout << '\n';
            break;
        case EDGESLIST:
            for (int i = 0; i < graph.size(); i++){
                for (int j = 0; j < graph[i].size(); j++){
                    std::cout << graph[i][j] << "  ";
                }
                std::cout << '\n';
            }
            std::cout << '\n';
            break;
    }
}
```