[1. Граф](#_Toc509246528)

[1.1. Представление и обход графов. Основная терминология](#_Toc509246529)

[1.2. Представления графов. Список ребер](#_Toc509246530)

[1.3. Представления графов. Списки смежности](#_Toc509246531)

[1.4. Реализация простейших операций над графами, представленными списками смежности](#_Toc509246532)

[1.5. Представления графов. Ортогональные списки смежности](#_Toc509246533)

[1.6. Представления графов. Структуры Вирта](#_Toc509246534)

[1.7. Реализация простейших операций над графом, представленным структурой Вирта](#_Toc509246535)

[1.8. Пример программы, реализующей простейшие операции над графом, представленным структурой Вирта](#_Toc509246536)

[1.9. Модифицированные структуры Вирта](#_Toc509246537)

[1.10. Представление отношений](#_Toc509246538)

[1.11. Топологическая сортировка](#_Toc509246539)

[1.12. Первый пример использования топологической сортировки](#_Toc509246540)

[1.13. Второй пример использования топологической сортировки](#_Toc509246541)

[1.14. Третий пример использования топологической сортировки](#_Toc509246542)

[1.15. Четвертый пример использования топологической сортировки](#_Toc509246543)

[1.16. Понятие о методе PERT](#_Toc509246544)

[1.17. Представление грамматики](#_Toc509246545)

[1.18. Обход графов (общие сведения)](#_Toc509246546)

[1.19. Обход графов в глубину](#_Toc509246547)

[1.20. Обход графов в ширину](#_Toc509246548)

[1.21. Путь между фиксированными вершинами](#_Toc509246549)

[1.22. Эйлеровы пути и циклы](#_Toc509246550)

[1.23. Алгоритмы на графах. Кратчайшие пути между всеми парами вершин. Алгоритм Уоршалла](#_Toc509246551)

[1.24. Применение алгоритма Уоршалла. Вычисление длин кратчайших путей между вершинами](#_Toc509246552)

[1.25. Применение алгоритма Уоршалла. Отыскание компонент сильной связности](#_Toc509246553)

[1.26. Применение алгоритма Уоршалла. Определение рекурсивности подпрограммы](#_Toc509246554)

[1.27. Кратчайшие пути между всеми парами вершин. Контуры в ориентированных графах](#_Toc509246555)

[1.28. Связность. Вычисление компонент связности](#_Toc509246556)

[1.29. Связность. Нахождение компонент двусвязности](#_Toc509246557)

[1.30. Остовы. Построение остова](#_Toc509246558)

[1.31. Остовы. Построение остова наименьшей стоимости](#_Toc509246559)

[1.32. Остовы. Построение фундаментального множества циклов](#_Toc509246560)

[1.33. Алгоритмы с возвратом (общие сведения)](#_Toc509246561)

[1.34. Построение алгоритмов с возвратом](#_Toc509246562)

[1.35. Гамильтоновы циклы](#_Toc509246563)

[1.36. Клики](#_Toc509246564)

[1.37. Независимые множества вершин графа](#_Toc509246565)

[1.38. Раскраски](#_Toc509246566)

[1.39. Алгоритмы раскраски графа](#_Toc509246567)

[1.40. Изоморфизм](#_Toc509246568)

[2. Эволюция и революция](#_Toc509246569)

[2.41. Второй уровень](#_Toc509246570)

[2.41.1. Третий уровень](#_Toc509246571)

[2.41.1.1. Четвертый уровень](#_Toc509246572)

[2.41.1.1.1. Пятый уровень](#_Toc509246573)

1. Граф

## Представление и обход графов. Основная терминология

На этом шаге мы приведем ***основные термины, связанные с представлением и обходом графов***.

***Определение 1.***

Пусть **V** - непустое множество, **V2** - множество всех его двухэлементных подмножеств. Пара **(V,E)**, где **E** - произвольное подмножество **V2**, называется ***неориентированным графом***. Элементы множества **V** называются ***вершинами***, а элементы множества **E** - ребрами. Если **{v1,v2}** - ребро, то вершины **v1** и **v2** будем называть ***концами ребра***.

Будем использовать символы **|V|** и **|E|** для обозначения, соответственно, числа вершин и числа ребер в графе **(V,E)**.

***Определение 2.***

Пусть **V** - непустое множество, **V\*V** - его декартов квадрат. Пара **(V,A)**, где **A** - произвольное подмножество **V\*V**, называется ***ориентированным графом (орграфом)***. Элементы множества **V** называются ***вершинами***, а элементы множества **A** - **дугами (ориентированными ребрами)**. Если **(v1,v2)** - дуга, то вершины **v1** и **v2**называются ***началом дуги*** и ***концом дуги***, соответственно.

Хотя неориентированные и ориентированные графы - существенно различные объекты, в определенных случаях неориентированные графы можно рассматривать как ориентированные графы, в которых каждому ребру соответствует две противоположно ориентированные дуги.

***Определение 3.***

Пусть **G** и **H** - графы, **f** - взаимно однозначное отображение множества **V(G)** на множество **V(H)**, а **g** - взаимно однозначное отображение **E(G)** на **E(H)**. Обозначим **Q** упорядоченную пару **(f,g)**. Будем говорить, что **Q** есть ***изоморфное отображение (изоморфизм)*** графа **G** на граф **H**, если выполняется следующее условие: вершина **x**инцидентна ребру **A** в графе **G** тогда и только тогда, когда вершина **fx** инцидентна ребру **gA** в графе **H**.

Если такой изоморфизм **Q** существует, то будем говорить, что графы **G** и **H** изоморфны. Ясно, что необходимым условием изоморфизма является тот факт, что **|V(G)|=|V(H)|** и **|E(G)|=|E(H)|**.

Мы можем рассматривать **Q** как операцию, преобразующую граф **G** в граф **H**, и в соответствии с этим писать **QG=H**. Удобно также писать **Qv=fv** и **QA=gA** (для каждой вершины **v** и каждого ребра **A** графа **G**).

***Автоморфизмом графа* G** называется изоморфизм графа **G** на себя.

***Определение 4.***

***Простой путь в неориентированном графе*** - это последовательность смежных ребер **(v1,v2), (v2,v3),..., (vk-1,vk)**, таких, что все **vi** кроме, быть может, **v1** и **vk** различны. ***Простой путь в ориентированном графе*** - это последовательность смежных одинаково ориентированных дуг **(v1,v2), (v2,v3),..., (vk-1,vk)**, таких, что все **vi** , кроме, быть может, **v1** и **vk** различны. Число ребер, составляющих простой путь, называется ***длиной простого пути***.

Простой путь называется:

* ***в неориентированном графе - цепью***, а
* ***в ориентированном графе - путем***.

Простой путь, начинающийся и заканчивающийся в одной и той же вершине, называется:

* ***в неориентированном графе - циклом***, а
* ***в ориентированном графе - контуром***.

***Расстояние между вершинами*** **u** ***и*** **v** в связном неориентированном графе - это минимальная длина цепи между **u** и **v**.

***Определение 5.***

Неориентированный граф называется ***ациклическим***, если в нем отсутствуют циклы. Будем называть ориентированный граф ***бесконтурным***, если он не содержит контуров. В литературе для обозначения таких ориентированных графов употребляется также термин "ациклический ориентированный граф".

***Неориентированный граф является связным***, если любые две его вершины можно соединить цепью. ***Ориентированный граф связен***, если связен неориентированный граф, полученный из него путем удаления ориентации его ребер.

Если существует простой путь из вершины **u** в вершину **v**, то будем говорить, что вершина **v** ***достижима из вершины*** **u**.

***Ориентированный граф называется сильно связным***, если любые его вершины взаимно достижимы, и ***односторонне связным***, если для любых двух вершин по крайней мере одна достижима из другой.

Существует один простой и важный тип графов, которому, однако, разные авторы дали одинаковое название, - это ***деревья***. Деревья важны не только потому, что они находят приложения в различных областях знания, но и в силу их особого положения в самой теории графов. Последнее вызвано предельной простотой структуры деревьев. Часто при решении какой-либо задачи о графах ее сначала исследуют на деревьях.

***Определение 6.***

***Дерево - это связный ациклический граф***. Если выбрана некоторая вершина **a0**, то она называется ***корнем дерева***, а само дерево - ***деревом с корнем***.

Всякий граф, не содержащий циклов, называется ***лесом***. Таким образом, компонентами леса являются деревья.

Известны и другие определения дерева. В приведенной ниже теореме отражены некоторые из них.

Назовем два ребра ***смежными***, если у них есть общая вершина.

Пусть **p** - натуральное число. Граф **Kp** будем называть ***полным***, если каждая пара его вершин смежна.

Пусть графы **G1** и **G2** имеют непересекающиеся множества вершин **V1** и **V2** и непересекающиеся множества ребер **E1** и ***E2***. Объединением **G1UG2** таких графов называется граф, множеством вершин которого является**V=V1UV2**, а множеством ребер - **E=E1UE2**.

***Теорема.***

Для графа **G**, имеющего **p** вершин и **q** ребер, следующие утверждения эквивалентны:

* **G** - дерево;
* любые две вершины в **G** соединены единственной цепью;
* **G** - связный граф и **p=q+1**;
* **G** - ациклический граф и **p=q+1**;
* **G** - ациклический граф, и если любую пару несмежных вершин соединить ребром **x**, то в графе **G+x**будет точно один цикл;
* **G** - связный граф, отличный от **Kp** для **p>=3**, и если любую пару несмежных вершин соединить ребром **x**, то в графе **G+x** будет точно один цикл;
* **G** - граф, отличный от **K3UK1** и **K3UK2, p=q+1**, и если любую пару несмежных вершин соединить ребром **x**, то в графе **G+x** будет точно один цикл.

Со следующего шага мы начнем рассматривать ***способы представления графов***.

## Представления графов. Список ребер

На этом шаге мы познакомимся с первым способом представления графов - ***списком ребер***.

Знакомство со способами представления и обработки графов весьма поучительно. С одной стороны, графы являются достаточно наглядными объектами. С другой стороны, машинное представление графов допускает большое разнообразие. Сложность получения ответа на тот или иной вопрос относительно данного графа зависит, естественно, от способа представления графа. Поэтому в алгоритмах на графах взаимосвязь "алгоритм + структура данных" проявляется очень сильно. Один и тот же алгоритм, реализованный на различных структурах данных, очень часто приводит к совершенно разным программам.

Во многих задачах на графах выбор представления является решающим для повышения эффективности алгоритма. С другой стороны, переход от одного представления к другому относительно прост и может быть выполнен за **O(|V|2)**операций [1, с.355]. Поэтому если решение задачи на графе обязательно требует числа операций, по крайней мере пропорционального **|V|2**, то время ее решения не зависит от способа представления графа, так как оно может быть изменено за **O(|V|2)** операций.

Более экономным в отношении памяти (особенно в случае так называемых неплотных графов, когда **|E|** гораздо меньше **|V|2**) по сравнению с матрицей смежностей является метод представления графа с помощью ***структуры смежности***, которая является в простейшем случае списком пар, соответствующих его ребрам [1, с.354].

Пара **<x,y>**, входящая в список ребер, соответствует ребру **{x,y}** в случае неориентированного графа и дуге **(x,y)**, если граф ориентированный.

Например, приведем списки ребер, соответствующие неориентированным графам:

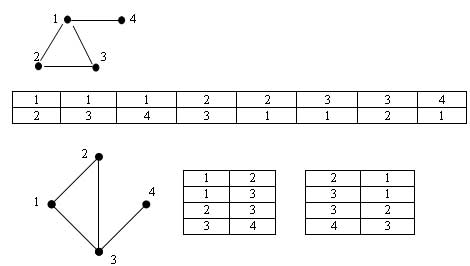


Рис.1. Примеры графов и списков ребер

Очевидно, что требуемый объем памяти в этом случае составляет **2|E|**. Неудобством является большое число шагов (порядка **|E|** в худшем случае), необходимое для получения множества вершин, смежных данной вершине. Ясно, что при представлении графа в виде списка ребер, информация о вершинах может оказаться труднодоступной. Так будет в случае, когда число ребер намного больше числа вершин. Ситуацию можно значительно улучшить, если упорядочить множество пар лексикографически и применять при поиске нужного ребра (дуги) двоичный поиск, но лучшим решением во многих случаях оказывается структура данных, которая называется ***списками смежности***.

(1) Рейнгольд Э., Нивергельт Ю., Део H. Комбинаторные алгоритмы. Теория и практика. - М.: Мир, 1980. - 476 с.

На следующем шаге мы рассмотрим ***списки смежности***.

## Представления графов. Списки смежности

На этом шаге мы рассмотрим ***представление графа с помощью списков смежности***.

Представление графа с помощью матрицы смежностей зачастую неудобно, поскольку количество вершин требуется знать заранее. Если граф должен создаваться или изменяться во время исполнения программы, то для каждого добавления или удаления вершины надо строить новую матрицу. Кроме того, даже если граф содержит малое число ребер (дуг) и матрица смежностей состоит в основном из нулей, память должна быть отведена для всех возможных дуг вне зависимости от того, существуют ли они. Если граф содержит **n** вершин, то должна быть отведена память для **n2**элементов.

Как и следовало ожидать, возможное решение - для представления графа использовать динамическую (связанную) структуру.

Укажем вначале на примере самый очевидный (в силу чего громоздкий и практически никогда не применяемый) способ представления графов с помощью ***динамических структур***.

Например, если ориентированный граф имеет следующий вид:

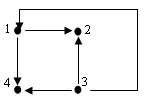


Рис.1. Пример графа

то без всяких ухищрений представим его в памяти следующим образом (символ **N** обозначает **NULL**):

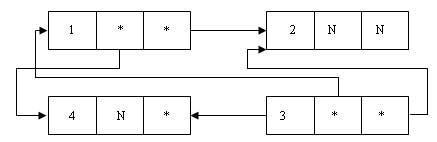


Рис.2. Представление графа

Совершенно ясно, что посредством ссылок можно выразить все связи в структуре, моделирующей ориентированный граф: вершинам графа соответствуют сами динамические переменные, а ребрам - ссылки.

Однако, так как в графе может существовать дуга между любыми двумя вершинами, то необходимо заранее знать, какое количество указателей резервировать для динамической переменной, а после построения может оказаться, что в структуре содержится очень много нулевых указателей (содержащих **NULL**).

Поэтому мы пойдем другим путем, и первым нашим шагом будет изучение классической структуры, называемой ***списками смежности***.

Список смежности содержит для каждой вершины **v**, принадлежащей **V**, список смежных ей вершин. Используя терминологию языка **C++**, можно утверждать, что каждый элемент такого списка является записью **R**, содержащей в поле **(\*R).Key** вершину графа, а в поле **(\*R).Sled** - указатель на следующую запись в списке; ясно, что для последней записи в списке **(\*R).Sled** содержит **NULL**.

Обозначим **beg[v]** - указатель на начало списка, содержащего вершины, смежные с вершиной **v**.

Изобразим схематически представление неориентированного графа с помощью списков смежности (символ **N**обозначает **NULL**):

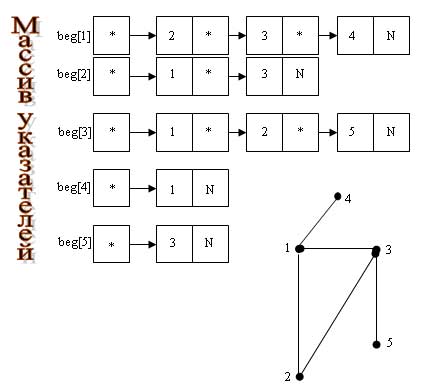


Рис.3. Представление неориентированного графа

Отметим, что для неориентированных графов каждое ребро представлено в списках смежности дважды.

Число ячеек памяти, необходимое для представления графа с помощью списков смежности, будет, очевидно, иметь порядок **|V|+|E|**.

На следующем шаге мы рассмотрим ***простейшие операции над графами, представленными списками смежности***.

## Реализация простейших операций над графами, представленными списками смежности

На этом шаге мы рассмотрим ***несколько операций над графами, представленными списками смежности***.

Теперь приведем реализацию на языке **C++** простейших операций над графами с использованием представления графов списками смежности.

Вначале опишем типы данных:

**#define** N 12; **// Количество вершин графа.**

**typedef** **struct** zveno \*svqz;

**typedef** **struct** zveno

{

**int** Key; **//Вершина графа.**

svqz Sled; **// Указатель на следующую смежную вершину.**

} Leader;

svqz beg[N]; **// Описание типа списков смежности.**

***1. Построение списков смежности, соответствующих данному ориентированному графу.*** Перед первым обращением к функции **MakeGraph** (создание графа) необходима инициализация списков смежности:

**for** (i=0;i<N;i++) beg[i] = **NULL**;

**void** MakeGraph (svqz beg[N])

**// Построение списков смежности beg графа.**

{

**int** x,y;

svqz ukzv,uzel; **//Рабочие указатели.**

cout<<"Вводите начало дуги: ";

cin>>x;

**while** (x!=0)

{

cout<< "Вводите конец дуги: ";

cin>>y;

AddGraph (x,y,beg);

cout<< "Вводите начало дуги: "; cin>>x;

}

}

***2. Вывод содержимого списков смежности, соответствующих ориентированному графу.***

**void** PrintGraph (svqz beg[N])

{

**int** i;

svqz ukzv; **//Рабочий указатель.**

**for** (i=1;i<N;i++)

{

cout<<i<<" ...";

ukzv = beg[i];

**if** (ukzv==**NULL**) cout<<"Пустой список!\n";

**else** {

**while** (ukzv!=**NULL**)

{ cout<< (\*ukzv).Key; ukzv = (\*ukzv).Sled; }

cout<<endl; }

}

}

Теперь рассмотрим реализацию унарных операций [1, с.22] на графе.

***3. Добавление дуги* (x,y) *(если ее не было!) к спискам смежности, соответствующим ориентированному графу.***

**void** AddGraph (**int** x, **int** y, svqz beg[N])

{

svqz ukzv,uzel; **//Рабочие указатели.**

**if** (beg[x]!=**NULL**)

{

Poisk (beg[x],y,&ukzv);

**if** (ukzv==**NULL**)

{ **// Добавление элемента в конец списка,**

**// заданного указателем beg[x].**

uzel = **new** (Leader);

(\*uzel).Key = y; (\*uzel).Sled = **NULL**; ukzv = beg[x];

**while** ((\*ukzv).Sled!=**NULL**)

ukzv = (\*ukzv).Sled;

(\*ukzv).Sled = uzel;

}

}

**else**

{

beg[x] = **new** (zveno);

(\*beg[x]).Key = y; (\*beg[x]).Sled = **NULL**;

}

}

***4. Удаление дуги между двумя заданными вершинами графа, представленного списками смежности (заметим, что вершины, инцидентные дуге, из графа не удаляются).***

**void** DeleteGraph (**int** x, **int** y, svqz beg[N])

{

svqz ukzv;

**if** (beg[x]!=**NULL**)

**//Удаление звена из списка без заглавного звена.**

{ **//Вершины в графе есть.**

Poisk (beg[x],y,&ukzv);

**if** (ukzv!=**NULL**) Udalenie (&beg[x],&ukzv);

**else** cout<<"Такой дуги в графе нет!\n";

}

**else** cout<<"Список пуст!\n";

}

**void** Poisk (svqz uksp, **int** ment, svqz \*res)

**// Поиск звена с информационным полем ment в**

**// однонаправленном списке uksp. \*res - указатель**

**// на найденное звено или NULL.**

{

svqz q;

\*res = **NULL**; q = uksp;

**while** ((q!=**NULL**)&&(\*res==**NULL**))

{ **if** ((\*q).Key==ment) \*res = q;

q = (\*q).Sled; }

}

**void** Udalenie (svqz \*ukstr, svqz \*zv)

**// Удаление звена, на которое ссылается указатель \*zv,**

**// из однонаправленного списка, заданного указателем \*ukstr.**

{

svqz ukzv,z;

**if** (((\*\*ukstr).Sled==**NULL**)&&(\*zv==\*ukstr))

**// В списке - один элемент!**

{ \*ukstr = **NULL**; **delete** zv; }

**else**

**if** (\*zv==\*ukstr) **// Удаляемый элемент - первый.**

{ \*ukstr = (\*\*ukstr).Sled; **delete** zv; }

**else** {

z = \*ukstr; ukzv = (\*\*ukstr).Sled;

**while** (ukzv!=\*zv)

{ z = ukzv; ukzv = (\*ukzv).Sled; }

(\*z).Sled = (\*(\*zv)).Sled; **delete** zv;

}

}

Отметим, что удаление вершины **v** из графа **G** приводит к подграфу, содержащему все вершины графа **G** за исключением **v**, и все ребра графа **G**, не инцидентные **v**. Заметим, что при данном представлении графа удаление вершин становится достаточно громоздким.

Приведем программу, демонстрирующую работу перечисленных выше функций.

Пример. Построение списков смежности, соответствующих ориентированному графу, вывод их на экран, добавление и удаление дуг.

**#include** <iostream.h>

**#define** N 12 **//Количество вершин графа.**

**#define** TRUE 1

**#define** FALSE 0

**typedef** **struct** zveno \*svqz;

**typedef** **struct** zveno

{

**int** Key; **//Вершина графа.**

svqz Sled; **//Указатель на следующую смежную вершину.**

} Leader;

**class** Spisok {

**private**:

svqz beg[N]; **//Описание типа списков смежности.**

svqz res; **//Указатель на найденное звено.**

**void** Poisk (svqz,**int**);

**void** Udalenie (svqz \*);

**public**:

Spisok ();

svqz GetPoisk () { **return** res; }

**void** MakeGraph ();

**void** AddGraph (**int**,**int**);

**void** DeleteGraph (**int**,**int**);

**void** PrintGraph ();

};

**void** main ()

{

Spisok A;

**int** x; **// Начало дуги.**

**int** y; **// Конец дуги.**

A.MakeGraph (); **// Построение списков смежности.**

**// Вывод списков смежностей.**

cout<<"Представление графа списками смежности\n";

A.PrintGraph (); cout<<endl;

**// Добавление дуги к графу.**

cout<<"Добавим к графу новую дугу...\n";

cout<<"Введите начало дуги: "; cin>>x;

cout<<"Введите конец дуги: "; cin>>y;

A.AddGraph (x,y);

cout<<"Представление графа списками смежности\n";

A.PrintGraph (); cout<<endl;

**//Удаление дуги из графа.**

cout<<"Удалим из графа заданную дугу...\n";

cout<<"Введите начало дуги: "; cin>>x;

cout<<"Введите конец дуги: "; cin>>y;

A.DeleteGraph (x,y);

cout<<"Представление графа списками смежности\n";

A.PrintGraph (); cout<<endl;

}

**void** Spisok::Poisk (svqz uksp,**int** ment)

**// Поиск звена с информационным полем ment в**

**// однонаправленном списке uksp. res - указатель**

**// на найденное звено или NULL.**

{

svqz q;

res = **NULL**; q = uksp;

**while** ((q!=**NULL**)&&(res==**NULL**))

{ **if** ((\*q).Key==ment) res = q; q = (\*q).Sled; }

}

**void** Spisok::AddGraph (**int** x,**int** y)

**// Добавление дуги (x,y) (если ее не было!) в граф,**

**// представленный списками смежности beg .**

{

svqz ukzv,uzel; **// Рабочие указатели.**

**if** (beg[x]!=**NULL**)

{

Poisk (beg[x],y);

**if** (GetPoisk()==**NULL**)

{ **//Добавление элемента в конец списка, заданного указателем beg[x].**

uzel = **new** (Leader);

(\*uzel).Key = y; (\*uzel).Sled = **NULL**; ukzv = beg[x];

**while** ((\*ukzv).Sled!=**NULL**)

ukzv = (\*ukzv).Sled;

(\*ukzv).Sled = uzel; }

}

**else** { beg[x] = **new** (zveno);(\*beg[x]).Key = y; (\*beg[x]).Sled = **NULL**; }

}

**void** Spisok::MakeGraph ()

**// Построение списков смежности beg графа.**

{

**int** x,y;

cout<<"Вводите начало дуги: "; cin>>x;

cout<<"Вводите конец дуги: "; cin>>y;

**while** (x!=0)

{

AddGraph (x,y);

cout<< "Вводите начало дуги: "; cin>>x;

cout<<"Вводите конец дуги: "; cin>>y;

}

}

**void** Spisok::Udalenie (svqz \*ukstr)

**//\* Удаление звена, на которое ссылается указатель res,**

**// из однонаправленного списка, заданного указателем \*ukstr**

{

svqz ukzv,z;

**if** (((\*\*ukstr).Sled==**NULL**)&&(res==\*ukstr)) **//В списке - один элемент!**

{ \*ukstr = **NULL**; **delete** res; }

**else** **if** (res==\*ukstr) **// Удаляемый элемент - первый.**

{ \*ukstr = (\*\*ukstr).Sled; **delete** res; }

**else** {

z = \*ukstr; ukzv = (\*\*ukstr).Sled;

**while** (ukzv!=res) { z = ukzv; ukzv = (\*ukzv).Sled; }

(\*z).Sled = (\*res).Sled; **delete** res; }

}

**void** Spisok::DeleteGraph (**int** x,**int** y)

**// Удаление дуги (x,y) из списков смежности beg.**

{

**if** (beg[x]!=**NULL**)

**// Удаление звена из списка без заглавного звена.**

{ **// Вершины в графе есть.**

Poisk (beg[x],y);

**if** (GetPoisk()!=**NULL**) Udalenie (&beg[x]);

**else** cout<<"Такой дуги в графе нет!\n"; }

**else** cout<<"Список пуст!\n";

}

**void** Spisok::PrintGraph ()

{

svqz ukzv; **// Рабочий указатель.**

**for** (**int** i=1;i<N;i++)

{ cout<<"..."<<i; ukzv = beg[i];

**if** (ukzv==**NULL**) cout<<" Пустой список!\n";

**else** {

**while** (ukzv!=**NULL**) { cout<<(\*ukzv).Key; ukzv = (\*ukzv).Sled; }

cout<<endl; }

}

}

Spisok::Spisok() { **for** (**int** i=0;i<N;i++) beg[i] = **NULL**; }

Текст этой программы можно взять [здесь](http://khpi-iip.mipk.kharkiv.edu/library/datastr/book_sod/kgsu/din81_1.zip).

Во многих алгоритмах необходимо динамически модифицировать структуру неориентированного графа путем добавления и удаления ребер. Если это нужно выполнить достаточно быстро, то полагаем, что в списках смежности элемент списка **beg[u]**, содержащий вершину **v**, снабжен указателем на элемент списка **beg[v]**, содержащий вершину **u**(и наоборот!), что схематически можно изобразить так:

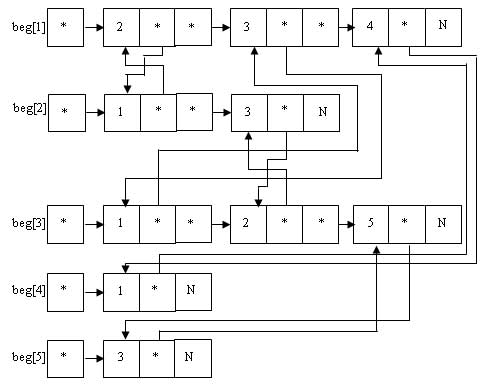


Рис.1. Представление графа

Тогда, удаляя "часть" ребра **(x,y)** из списка, мы можем легко за число шагов, ограниченное константой, удалить другую "часть" ребра, а именно: **(y,x)**, не просматривая весь список **beg[y]**.

Далее, можно еще более усложнить структуру, если предположить, что каждый список смежных вершин является двунаправленным (попробуйте изобразить это схематически!).

***Замечания****.*

1. *Аналогичным образом можно определить для каждой вершины* ***v*** *ориентированного графа список, определенный указателем* ***SUC[v]****, содержащий вершины из множества* ***{u: u->v}****. "Еще один способ состоит в том, что для каждой вершины выписываются номера следующих, не смежных с ней..." [2, с.32].*
2. *Представление графа в виде списков смежности определяет "порядок" ребер, смежных некоторой вершине графа. Граф с подобным упорядочиванием ребер называют* ***упорядоченным графом****. Дадим формальное определение.*

***Определение 1 [3, с.237].***

*Множество* ***V={v1,v2,...,vn}*** *вершин вместе с множеством* ***{Lv1, Lv2,...,Lvn}*** *упорядоченных списков упорядоченных пар вершин называют* ***упорядоченным графом****.*

*Упорядоченный граф определяет единственный неупорядоченный граф. Обратное утверждение неверно, поскольку в общем случае возможно много способов упорядочения графа.*

***Определение 2 [3, с.248].***

***Упорядоченным ориентированным графом*** *называется пара* ***G=(V,E)****, где* ***V*** *- конечное множество вершин, а* ***E*** *- множество упорядоченных списков дуг. Элементы* ***E*** *имеют вид Lv=((v,w1),(v,w2),...,(v,wk)), где v,wi принадлежат* ***V****.*

(1) Свами М., Тхуласираман К. Графы, сети и алгоритмы. - М.: Мир, 1984. - 454 с.

(2) Зыков А.А. Основы теории графов. М.: Наука, 1987. - 384 с.

(3) Кук Д., Бейз Г. Компьютерная математика. - М.: Наука, 1990. - 384 с.

На следующем шаге мы рассмотрим ***представление графов с помощью ортогональных списков смежности***.

## Представления графов. Ортогональные списки смежности

На этом шаге мы рассмотрим ***представление графов с помощью орогональных списков смежности***.

На рисунке, приведенном ниже, изображен граф и его представление с помощью структуры данных, которую мы будем называть ***ортогональными списками смежности***.

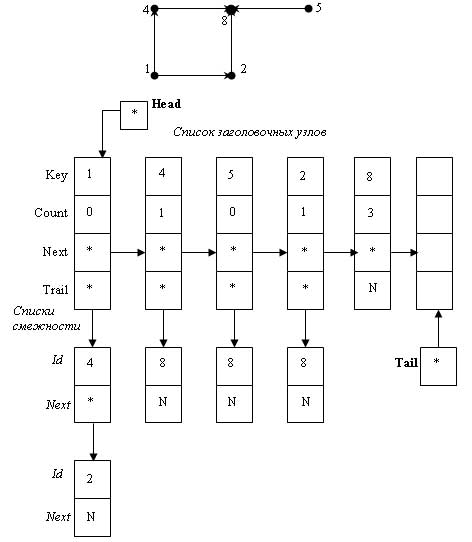


Рис.1.Ортогональные списки смежности

Вершины графа находятся в связанном списке, который называется ***списком заголовочных узлов***. Каждый узел этого списка содержит четыре поля.

Если указатель **P** указывает на заголовочный узел, представляющий вершину **Q** графа, то:

* поле **(\*P).Key** содержит информацию, связанную с этой вершиной **Q**;
* поле **(\*P).Count** содержит количество вершин, "предшествующих" данной;
* поле **(\*P).Next** содержит указатель на заголовочный узел, представляющий следующую вершину графа (если такая вершина есть) в списке заголовочных узлов;
* каждый заголовочный узел является заглавным звеном списка узлов второго типа, называемых ***дуговыми узлами***. Такой список мы также будем называть ***списком смежности***. Каждый узел списка смежности представляет конечную вершину некоторой дуги графа. Поле **(\*P).Trail** указывает на список смежности, представляющий дуги, выходящие из вершины **Q** графа (английское слово **"Trail"** переводится как ***"тащиться, свисать, волочиться"***).

Каждый узел списка смежности содержит два поля: **Id** и **Next**, причем если **Q** указывает на списочный узел, представляющий дугу **(A,B)**, то:

* поле **(\*Q).Id** содержит вершину **B** графа;
* поле **(\*Q).Next** указывает на дуговой узел, представляющий следующую дугу, выходящую из вершины **A** графа (если такая дуга есть).

На рисунке 1 показан граф и представление его ортогональными списками смежности. Каждый дуговой узел содержится в единственном списке смежности, представляющем все дуги, выходящие из данной вершины графа. Термин ***распределенные узлы*** используется как для заголовочных, так и для дуговых узлов структуры со многими связями, представляющей граф.

Приведем реализацию на языке **C++** простейших операций над графами с использованием представления графов в виде ортогональных списков смежности.

Вначале опишем типы данных:

**typedef** **struct** L \*Lref; **// Тип: указатель на заголовочный узел.**

**typedef** **struct** T \*Tref; **// Тип: указатель на дуговой узел.**

**//Описание типа заголовочного узла.**

**typedef** **struct** L

{

**int** Key; **// Имя заголовочного узла.**

**int** Count; **//Количество предшественников.**

Tref Trail; **//Указатель на список смежности.**

Lref Next; **//Указатель на следующий узел в списке заголовочных узлов.**

} Leader;

**typedef** **struct** T **//Описание типа дугового узла.**

{

**int** Id;

Tref Next;

} Trailer;

Приведем программу, демонстрирующую работу функций, предназначенных для выполнения некоторых операций над графами.

Пример. Построение ортогональных списков смежности, соответствующих ориентированному графу, вывод на экран структуры ортогональных списков смежности, добавление и удаление дуг, добавление и удаление вершин.

**#include** <iostream.h>

**#define** TRUE 1

**#define** FALSE 0

**typedef** **int** Boolean;

**typedef** **struct** L \*Lref; **//Тип: указатель на заголовочный узел.**

**typedef** **struct** T \*Tref; **//Тип: указатель на дуговой узел.**

**//Описание типа заголовочного узла.**

**typedef** **struct** L

{ **int** Key; **//Имя заголовочного узла.**

**int** Count; **//Количество предшественников.**

Tref Trail; **//Указатель на список смежности.**

Lref Next; **//Указатель на следующий узел в списке заголовочных узлов.**

} Leader;

**typedef** **struct** T **//Описание типа дугового узла.**

{

**int** Id;

Tref Next;

} Trailer;

**class** Spisok

{

**private**:

Lref Head; **//Указатель на начало списка заголовочных узлов.**

Lref Tail; **//Указатель на фиктивный узел**

**// в конце списка заголовочных узлов.**

**void** SearchGraph (**int**, Lref \*);

**void** Search (**int**, Lref \*);

**public**:

Spisok () {**//Инициализация списка заголовочных узлов.**

Head = Tail = **new** (Leader); }

**void** MakeGraph ();

**void** AddGraph (**int**, **int**);

**void** DeleteGraph (**int**, **int**);

**void** PrintGraph ();

**void** DeleteY (**int**);

};

**void** main ()

{

Spisok A;

**int** x,y; **//Начало и конец дуги**

**//Построение и вывод структуры ортогональных списков.**

A.MakeGraph ();

A.PrintGraph (); cout<<endl;

**//Добавление дуги к графу.**

cout<<"Добавим к графу новую дугу...\n";

cout<<"Введите начало дуги: "; cin>>x;

cout<<"Введите конец дуги: "; cin>>y;

A.AddGraph (x,y);

A.PrintGraph (); cout<<endl;

**//Удаление дуги из графа.**

cout<<"Удалим из графа заданную дугу...\n";

cout<<"Введите начало дуги: "; cin>>x;

cout<<"Введите конец дуги: "; cin>>y;

A.DeleteGraph (x,y);

A.PrintGraph (); cout<<endl;

**//Удаление вершины графа.**

cout<< "Введите удаляемую вершину: "; cin>>y;

A.DeleteY (y);

A.PrintGraph (); cout<<endl;

}

**void** Spisok::Search (**int** w, Lref \*h)

**//Функция возвращает в \*h указатель на заголовочный узел**

**//ключом w. Если узел отсутствует, то функция возвращает NULL .**

{

\*h = Head;

(\*Tail).Key = w; **//Поиск "с барьером".**

**while** ((\*\*h).Key!=w) \*h = (\*\*h).Next;

**if** (\*h==Tail) **//В списке заголовочных узлов нет узла с ключом w.**

\*h = **NULL**;

}

**void** Spisok::SearchGraph (**int** w,Lref \*h)

**//Функция возвращает в \*h указатель на заголовочный узел**

**//с ключом w. Если заголовочный узел отсутствует, то он**

**//добавляется в список.**

{

\*h = Head; (\*Tail).Key = w;

**while** ((\*\*h).Key!=w) \*h = (\*\*h).Next;

**if** (\*h==Tail)

**//В списке заголовочных узлов нет узла с ключом w.**

**//Поместим его в конец списка \*Head.**

{ Tail = **new** (Leader); (\*\*h).Count = 0;

(\*\*h).Trail = **NULL**; (\*\*h).Next = Tail; }

}

**void** Spisok::MakeGraph ()

**//Функция возвращает указатель Head на ортогональные**

**//списки смежности, соответствующую ориентированному графу.**

{

**int** x,y;

Lref p,q; **//Рабочие указатели.**

Tref t,r; **// Рабочие указатели.**

Boolean Res; **//Флаг наличия дуги.**

cout<<"Вводите начальную вершину дуги: ";

cin>>x;

**while** (x!=0)

{

cout<<"Вводите конечную вершину дуги: "; cin>>y;

**//Определим, существует ли в графе дуга (x,y)?**

SearchGraph (x,&p); SearchGraph (y,&q);

r = (\*p).Trail; Res = FALSE;

**while** ((r!=**NULL**)&&(!Res))

**if** ((\*r).Id==y) Res = TRUE;

**else** r = (\*r).Next;

**if** (!Res) **//Если дуга отсутствует, то поместим её в граф.**

{ t = **new** (Trailer); (\*t).Id = y;

(\*t).Next = (\*p).Trail; (\*p).Trail = t; (\*q).Count++; }

cout<<"Вводите начальную вершину дуги: "; cin>>x;

}

}

**void** Spisok::AddGraph (**int** x,**int** y)

**//Добавление дуги (x,y) (если ее не было!) к связанной**

**//структуре, соответствующей ориентированному графу Head.**

{

Lref p,q; **//Рабочие указатели.**

Tref t,r; **//Рабочие указатели.**

Boolean Res; **//Флаг наличия в графе данной дуги.**

**//Определим, существует ли в графе дуга (x,y)?**

SearchGraph (x,&p);

SearchGraph (y,&q);

r = (\*p).Trail; Res = FALSE;

**while** ((r!=**NULL**)&&(!Res))

**if** ((\*r).Id==y) Res = TRUE; **else** r = (\*r).Next;

**if** (!Res)

{ **//Если дуга отсутствует, то поместим её в граф.**

t = **new** (Trailer); (\*t).Id = y; (\*t).Next = (\*p).Trail;

(\*p).Trail = t; (\*q).Count++;

}

}

**void** Spisok::DeleteGraph (**int** x,**int** y)

**//Функция возвращает указатель Head на связанную**

**//структуру, соответствующую ориентированному графу**

**//и полученную удалением дуги (x,y).**

{

Lref p,q; **//Рабочие указатели.**

Tref t,r; **//Рабочие указатели.**

Boolean Res; **//Флаг наличия в графе данной дуги.**

**//Определим, существует ли в графе дуга (x,y)?**

Search (x, &p); Search (y, &q);

**if** ((p!=**NULL**)&&(q!=**NULL**))

{ **//Вершины x и y в графе есть.**

r = (\*p).Trail; Res = FALSE;

**while** ((r!=**NULL**)&&(!Res))

**if** ((\*r).Id==y) Res = TRUE;

**else** r = (\*r).Next;

**if** (Res) **//Если дуга существует, то удалим её.**

**if** (r==(\*p).Trail)

{ (\*p).Trail = (\*(\*p).Trail).Next; **delete** r; (\*q).Count--; }

**else**

{

t = (\*p).Trail;

**while** ((\*t).Next!=r) t = (\*t).Next;

(\*t).Next = (\*(\*t).Next).Next; **delete** r; (\*q).Count--; }

}

}

**void** Spisok::PrintGraph ()

**//Вывод структуры смежности, заданной указателем**

**//Head и соответствующей ориентированному графу.**

{

Lref p; **//Рабочий указатель.**

Tref q; **//Рабочий указатель.**

p = Head;

**while** (p!=Tail)

{

cout<<"("<<(\*p).Key; q = (\*p).Trail;

**while** (q!=**NULL**)

{ cout<<(\*q).Id; q = (\*q).Next; }

cout<<")"; p = (\*p).Next; cout<<" ";

}

}

**void** Spisok::DeleteY (**int** y)

**//Функция возвращает указатель Head на связанную струк-**

**//туру, соответствующую графу с удаленной вершиной y.**

{

Lref p,q; **//Рабочие указатели.**

Tref r,s; **//Рабочие указатели.**

**int** x; **//Рабочая переменная.**

**//Удаление всех дуг (x,y), оканчивающихся в вершине y.**

p = Head;

**while** (p!=Tail)

{ x = (\*p).Key; DeleteGraph (x,y); p = (\*p).Next; }

**//Удаление списка смежности вершины y.**

SearchGraph (y, &p); r = (\*p).Trail;

**while** (r!=**NULL**)

{ s = r; r = (\*r).Next; **delete** s; }

**//Удаление узла, содержащего вершину y, из списка заголовочных узлов.**

q = Head;

**if** (q==p) { Head = (\*Head).Next; **delete** q; }

**else** {

**while** ((\*q).Next!=p) q = (\*q).Next;

(\*q).Next = (\*p).Next; **delete** p; }

}

Текст этой программы можно взять [здесь](http://khpi-iip.mipk.kharkiv.edu/library/datastr/book_sod/kgsu/din82_1.zip).

На следующем шаге мы рассмотрим ***представление графов с помощью структур Вирта***.

## Представления графов. Структуры Вирта

На этом шаге мы рассмотрим ***представление графов с помощью структуры Вирта***.

Используем для представления графа в памяти ЭВМ подход Никлауса Вирта [1, с.211-219], который предполагает построение динамической структуры со многими связями. Вершины графа представляются линейным списком, состоящим из заголовочных узлов. Каждый заголовочный узел содержит четыре поля.

Если указатель **P** указывает на заголовочный узел, представляющий вершину **Q** графа, то:

* поле **(\*P).Key** содержит информацию, связанную с вершиной **Q**;
* поле **(\*P).Count** содержит количество вершин, "предшествующих" данной;
* поле **(\*P).Next** содержит указатель на заголовочный узел, представляющий следующую вершину графа (если такая вершина есть) в списке заголовочных узлов;
* каждый заголовочный узел является заглавным звеном списка узлов второго типа, называемых ***дуговыми узлами***. Такой список мы будем называть ***списком смежности***. Поле **(\*P).Trail** указывает на список смежности, представляющий дуги, выходящие из вершины **Q** графа (английское слово **"Trail"** переводится как ***"тащиться, свисать, волочиться"***).

Каждый узел списка смежности содержит два поля: **Id** и **Next**, причем если **Q** указывает на дуговой узел, представляющий дугу **(A,B)**, то:

* поле **(\*Q).Id** указывает на заголовочный узел, представляющий вершину **B** графа;
* поле **(\*Q).Next** указывает на дуговой узел, представляющий следующую дугу, выходящую из вершины **A** графа (если такая дуга есть).

Каждый дуговой узел содержится в единственном списке смежности, представляющем все дуги, выходящие из данной вершины графа.

***Вывод***. Каждый заголовочный узел содержит поля **Key, Count** и два указателя:

* первый указывает на список смежных дуг, выходящих из вершины графа,
* второй - на следующий заголовочный узел в списке заголовочных узлов.

Каждый дуговой узел содержит два указателя:

* на следующий дуговой узел в списке смежности и
* на заголовочный узел, представляющий ту вершину графа, которая является концом дуги.

На рисунке показан ориентированный граф и его представление с помощью структуры Вирта (символ **N** обозначает **NULL**):

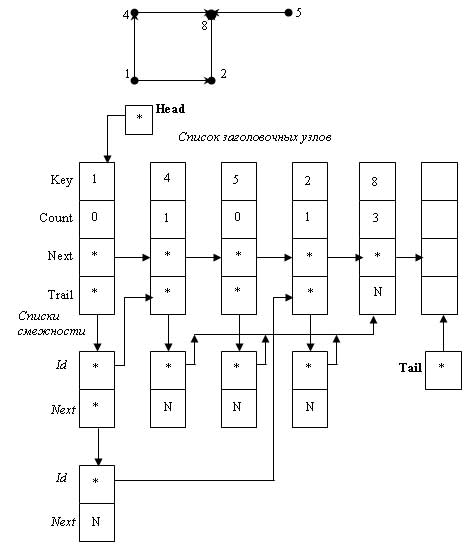


Рис.1. Представление графа с помощью структуры Вирта

(1)Вирт H. Алгоритмы + структуры данных = программы. - М.: Мир, 1985. - 406 с.

На следующем шаге мы рассмотрим ***реализацию простейших операций над графом, представленным структурой Вирта***.

## Реализация простейших операций над графом, представленным структурой Вирта

На этом шаге мы рассмотрим ***реализацию простейших операций над графом, представленным структурой Вирта*** .

Теперь приведем реализацию на языке **C++** простейших операций над графами с использованием структуры Вирта.

Вначале опишем типы данных:

**typedef** **struct** L \*Lref; **//Тип: указатель на заголовочный узел.**

**typedef** **struct** T \*Tref; **//Тип: указатель на дуговой узел.**

**//Описание типа заголовочного узла.**

**typedef** **struct** L

{

**int** Key; **//Имя заголовочного узла.**

**int** Count; **//Количество предшественников.**

Tref Trail; **//Указатель на список смежности.**

Lref Next; **//Указатель на следующий узел в**

**//списке заголовочных узлов.**

} Leader;

**//Описание типа дугового узла.**

**typedef** **struct** T

{

Lref Id;

Tref Next;

} Trailer;

***1. Поиск в структуре Вирта заголовочного узла с заданным значением поля* Key *и добавление его в список заголовочных узлов в случае отсутствия.***

Lref SearchGraph (**int** w, Lref Head, Lref \*Tail)

**//Функция возвращает указатель на заголовочный узел**

**//с ключом w. Если заголовочный узел отсутствует,**

**//то он добавляется в список. Head - указатель на**

**//структуру Вирта.**

{

Lref h;

h = Head; (\*\*Tail).Key = w;

**while** ((\*h).Key!=w) h = (\*h).Next;

**if** (h==\*Tail)

{**//В списке заголовочных узлов нет узла с ключом w.**

**//Поместим его в конец списка Head.**

\*Tail = **new** (Leader); (\*h).Count = 0;

(\*h).Trail = **NULL**; (\*h).Next = \*Tail;}

**return** h;

}

***2. Построение структуры Вирта, соответствующей данному ориентированному графу.*** Перед самым первым обращением к функции создания графа **MakeGraph ()** инициализируем список заголовочных узлов:

Head = **new** (Leader); Tail = Head;.

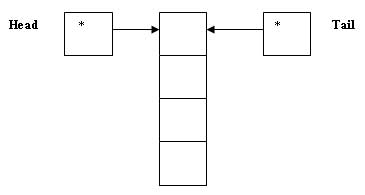


Рис.1. Результат инициализации

Добавление к графу новых вершин и инцидентных им ребер осуществляется обращением к функции **MakeGraph()**.

**void** MakeGraph (Lref \*Head, Lref \*Tail)

**//Функция возвращает указатель \*Head на структуру**

**// Вирта, соответствующую ориентированному графу.**

{

**int** x,y;

Lref p,q; **//Рабочие указатели.**

Tref t,r; **//Рабочие указатели.**

Boolean Res; **//Флаг наличия дуги.**

cout<<"Вводите начальную вершину дуги: ";

cin>>x;

**while** (x!=0)

{ cout>>"Вводите конечную вершину дуги: "; cin>>y;

**//Определим, существует ли в графе дуга (x,y)?**

p = SearchGraph (x,\*Head,Tail); q = SearchGraph (y,\*Head,Tail);

r = (\*p).Trail; Res = FALSE;

**while** ((r!=**NULL**) && (!Res))

**if** ((\*r).Id==q) Res = TRUE;

**else** r = (\*r).Next;

**if** (!Res) **//Если дуга не существует, то поместим её в граф.**

{ t = **new** (Trailer); (\*t).Id = q; (\*t).Next = (\*p).Trail;

(\*p).Trail = t; (\*q).Count++; }

cout<< "Вводите начальную вершину дуги: "; cin>>x; }

}

***3. Вывод структуры Вирта, соответствующей ориентированному графу.***

**void** PrintGraph (Lref Head, Lref Tail)

**//Вывод структуры Вирта, заданной указателем Head и**

**//соответствующей ориентированному графу.**

{

Lref p; **//Рабочий указатель.**

Tref q; **// Рабочий указатель.**

p = Head;

**while** (p!=Tail)

{ cout<<"("<<(\*p).Key; q = (\*p).Trail;

**while** (q!=**NULL**)

{cout<<(\*(\*q).Id).Key; q = (\*q).Next;}

cout<<")"; p = (\*p).Next; cout<<" "; }

}

Теперь рассмотрим реализацию унарных операций [1, с.22] на графе.

***4. Добавление дуги* (x,y) *(если ее не было) к структуре Вирта, соответствующей ориентированному графу.***

**void** AddGraph (**int** x,**int** y, Lref \*Head, Lref \*Tail)

**//Добавление дуги (x,y) (если ее не было!) к структуре**

**//Вирта, соответствующей ориентированному графу \*Head.**

{

Lref p,q; **//Рабочие указатели.**

Tref t,r; **// Рабочие указатели.**

Boolean Res; **//Флаг наличия в графе данной дуги.**

p = SearchGraph (x,\*Head,Tail); q = SearchGraph (y,\*Head,Tail);

**//Определим, существует ли в графе дуга (x,y)?**

r = (\*p).Trail; Res = FALSE;

**while** ((r!=**NULL**) && (!Res))

**if** ((\*r).Id==q) Res = TRUE;

**else** r = (\*r).Next;

**if** (!Res)

{ **//Если дуга не существует, то поместим её в граф.**

t = **new** (Trailer); (\*t).Id = q; (\*t).Next = (\*p).Trail;

(\*p).Trail = t; (\*q).Count++; }

}

***5. Удаление дуги между двумя вершинами структуры Вирта, определенной указателем* Head.** Заметим, что концевые вершины дуги из структуры Вирта (а значит, и из графа) не удаляются.

**void** DeleteGraph (**int** x,**int** y, Lref \*Head, Lref \*Tail)

**//Функция возвращает указатель \*Head на структуру**

**//Вирта, соответствующую ориентированному графу и**

**//полученную удалением дуги (x,y).**

{

Lref p,q; **//Рабочие указатели.**

Tref t,r; **//Рабочие указатели.**

Boolean Res; **//Флаг наличия дуги.**

**// Определим, существует ли в графе дуга (x,y)?**

p = Search (x,\*Head,\*Tail);q = Search (y,\*Head,\*Tail);

**if** ((p!=**NULL**) && (q!=**NULL**))

{ **// Вершины в графе есть.**

r = (\*p).Trail; Res = FALSE;

**while** ((r!=**NULL**) && (!Res))

**if** ((\*r).Id==q) Res = TRUE;

**else** r = (\*r).Next;

**if** (Res) **//Если дуга существует, то удалим её.**

**if** (r==(\*p).Trail)

{ (\*p).Trail = (\*(\*p).Trail).Next; **delete** r; (\*q).Count--; }

**else** {

t = (\*p).Trail;

**while** ((\*t).Next!=r) t = (\*t).Next;

(\*t).Next = (\*(\*t).Next).Next; **delete** r; (\*q).Count--; }

}

}

Lref Search (**int** w, Lref Head, Lref Tail)

**//Функция возвращает указатель на заголовочный узел с**

**//ключом w. Если узел отсутствует, то функция возвращает**

**//NULL.**

{

Lref h;

h = Head;

(\*Tail).Key = w; **//Поиск "с барьером".**

**while** ((\*h).Key!=w) h = (\*h).Next;

**if** (h==Tail) **//В списке заголовочных узлов нет узла с ключом w.**

h = **NULL**;

**return** h;

}

***6. Удаление вершины графа.*** Необходимо учесть, что удаление вершины **v** из графа **G** приводит к подграфу, содержащему все вершины графа **G** за исключением **v**, и все ребра графа **G**, не инцидентные **v**.

**void** DeleteY (**int** y, Lref Head, Lref Tail)

**//Функция возвращает указатель \*Head на структуру Вирта,**

**//соответствующую графу с удаленной вершиной y.**

{

Lref p,q; **//Рабочие указатели.**

Tref r,s; **// Рабочие указатели.**

**int** x; **// Рабочая переменная.**

**//Удаление всех дуг (x,y), оканчивающихся в вершине y.**

p = \*Head;

**while** (p!=\*Tail)

{ x = (\*p).Key; DeleteGraph (x,y,Head,Tail);p = (\*p).Next; }

**//Удаление списка смежности вершины y.**

p = SearchGraph (y,\*Head,Tail);

r = (\*p).Trail;

**while** (r!=**NULL**)

{ s = r; r = (\*r).Next; **delete** s; }

**//Удаление узла, содержащего вершину y, из**

**//списка заголовочных узлов.**

q = \*Head;

**if** (q==p)

{ \*Head = (\*\*Head).Next; **delete** q;}

**else** {

**while** ((\*q).Next!=p) q = (\*q).Next;

(\*q).Next = (\*p).Next; **delete** p; }

}

***7. Удаление структуры Вирта, соответствующей графу.*** Не вызывает трудностей написание фрагмента, осуществляющего эту операцию:

. . .

t = Head;

**while** (t!=Tail)

{ Free2Graph (&(\*t).Trail); t = (\*t).Next; }

Free1Graph (&Head,&Tail);

**delete** Tail;

. . .

Здесь мы воспользуемся рекурсивными функциями очистки динамической памяти, занятой линейным списком:

**void** Free1Graph (Lref \*Head, Lref \*Tail)

**//Очистка динамической памяти, занятой линейным списком,**

**//заданным указателем \*Head.**

{

**if** (\*Head!=\*Tail)

{ Free1Graph (&(\*\*Head).Next,Tail); **delete** \*Head; \*Head = **NULL**; }

}

**void** Free2Graph (Tref \*X)

**//Очистка динамической памяти, занятой линейным списком,**

**//заданным указателем \*X.**

{

**if** (\*X!=**NULL**)

{ Free2Graph (&(\*\*X).Next); **delete** \*X; \*X = **NULL**; }

}

(1) Свами М., Тхуласираман К. Графы, сети и алгоритмы. - М.: Мир, 1984. - 454 с.

На следующем шаге мы приведем ***пример программы, в которой реализованы рассмотренные операции над графом, представленным структурой Вирта***.

## Пример программы, реализующей простейшие операции над графом, представленным структурой Вирта

На этом шаге мы приведем ***пример программы, реализующей простейших операций над графом, представленным структурой Вирта***.

Приведем программу, демонстрирующую работу всех приведенных на [предыдущем шаге](http://khpi-iip.mipk.kharkiv.edu/library/datastr/book_sod/kgsu/din_0084.html) функций.

Пример. Построение динамической структуры Вирта, соответствующей ориентированному графу, вывод на экран его структуры, добавление и удаление ребер, добавление и удаление вершин.

**#include** <iostream.h>

**#define** TRUE 1

**#define** FALSE 0

**typedef** **int** Boolean;

**typedef** **struct** L \*Lref; **//Тип: указатель на заголовочный узел.**

**typedef** **struct** T \*Tref; **//Тип: указатель на дуговой узел.**

**//Описание типа заголовочного узла.**

**typedef** **struct** L

{

**int** Key; **//Имя заголовочного узла.**

**int** Count; **//Количество предшественников.**

Tref Trail; **//Указатель на список смежности.**

Lref Next; **//Указатель на следующий узел в**

**//списке заголовочных узлов.**

} Leader;

**//Описание типа дугового узла.**

**typedef** **struct** T

{

Lref Id;

Tref Next;

} Trailer;

**class** Spisok {

**private**:

Lref Head; **//Указатель на список заголовочных узлов.**

Lref Tail; **//Указатель на фиктивный элемент**

**//в конце списка заголовочных узлов.**

Lref SearchGraph(**int**);

Lref Search(**int**);

**public**:

Spisok () {**//Инициализация списка заголовочных узлов.**

Head = Tail = **new** (Leader); }

~Spisok(); **//Деструктор.**

**void** MakeGraph();

**void** AddGraph(**int**, **int**);

**void** DeleteGraph(**int**, **int**);

**void** PrintGraph();

**void** DeleteY(**int**);

**void** Free1Graph(Lref \*, Lref \*);

**void** Free2Graph(Tref \*);

};

**void** main ()

{

Spisok A;

**int** x,y; **//Начало и конец дуги.**

**//Построение и вывод структуры Вирта.**

A.MakeGraph ();

A.PrintGraph (); cout<<endl;

cout<< "Добавим к графу новую дугу...\n";

cout<< "Введите начало дуги: "; cin>>x;

cout<< "Введите конец дуги: "; cin>>y;

A.AddGraph (x,y);

A.PrintGraph (); cout<<endl;

cout<< "Добавим к графу новое ребро...\n";

cout<< "Введите первую концевую вершину ребра: "; cin>>x;

cout<< "Введите вторую концевую вершину ребра: "; cin>>y;

A.AddGraph (x,y);

A.AddGraph (y,x);

A.PrintGraph (); cout<<endl;

cout<< "Удалим из графа заданную дугу...\n";

cout<< "Введите начало дуги: "; cin>>x;

cout<< "Введите конец дуги: "; cin>>y;

A.DeleteGraph (x,y);

A.PrintGraph (); cout<<endl;

cout<< "Удалим из графа заданное ребро...\n";

cout<< "Введите первую концевую вершину ребра: "; cin>>x;

cout<< "Введите вторую концевую вершину ребра: "; cin>>y;

A.DeleteGraph (x,y);

A.DeleteGraph (y,x);

A.PrintGraph (); cout<<endl;

cout<< "Введите удаляемую вершину: "; cin>>y;

A.DeleteY (y);

A.PrintGraph (); cout<<endl;

}

Spisok::~Spisok()

{

**//Очистка структуры Вирта.**

Lref t = Head;

**while** (t!=Tail)

{ Free2Graph (&(\*t).Trail); t = (\*t).Next; }

Free1Graph (&Head,&Tail);

**delete** Tail;

}

Lref Spisok::SearchGraph (**int** w)

**//Функция возвращает указатель на заголовочный узел**

**//с ключом w. Если заголовочный узел отсутствует, то он**

**//добавляется в список. Head - указатель на структуру Вирта.**

{

Lref h;

h = Head; (\*Tail).Key = w;

**while** ((\*h).Key!=w) h = (\*h).Next;

**if** (h==Tail)

**//В списке заголовочных узлов нет узла с ключом w.**

**//Поместим его в конец списка Head.**

{ Tail = **new** (Leader); (\*h).Count = 0;

(\*h).Trail = **NULL**; (\*h).Next = Tail; }

**return** h;

}

Lref Spisok::Search (**int** w)

**//Функция возвращает указатель на заголовочный узел**

**//ключом w. Если узел отсутствует, то функция возвращает NULL .**

{

Lref h = Head;

(\*Tail).Key = w; **//Поиск "с барьером".**

**while** ((\*h).Key!=w) h = (\*h).Next;

**if** (h==Tail) **//В списке заголовочных узлов нет узла с ключом w.**

h = **NULL**;

**return** h;

}

**void** Spisok::MakeGraph ()

**//Функция возвращает указатель Head на структуру**

**//Вирта, соответствующую ориентированному графу.**

{

**int** x,y;

Lref p,q; **//Рабочие указатели.**

Tref t,r; **// Рабочие указатели.**

Boolean Res; **//Флаг наличия дуги.**

cout<<"Вводите начальную вершину дуги: ";

cin>>x;

**while** (x!=0)

{

cout<<"Вводите конечную вершину дуги: "; cin>>y;

**//Определим, существует ли в графе дуга (x,y)?**

p=SearchGraph (x); q=SearchGraph (y);

r = (\*p).Trail; Res = FALSE;

**while** ((r!=**NULL**)&&(!Res))

**if** ((\*r).Id==q) Res = TRUE;

**else** r = (\*r).Next;

**if** (!Res) **//Если дуга отсутствует, то поместим её в граф.**

{ t = **new** (Trailer); (\*t).Id = q;

(\*t).Next = (\*p).Trail; (\*p).Trail = t; (\*q).Count++; }

cout<<"Вводите начальную вершину дуги: "; cin>>x;

}

}

**void** Spisok::AddGraph (**int** x,**int** y)

**//Добавление дуги (x,y) (если ее не было!) к структуре**

**//Вирта, соответствующей ориентированному графу Head.**

{

Lref p,q; **//Рабочие указатели.**

Tref t,r; **//Рабочие указатели.**

Boolean Res; **//Флаг наличия в графе данной дуги.**

**//Определим, существует ли в графе дуга (x,y)?**

p=SearchGraph (x); q=SearchGraph (y);

r = (\*p).Trail; Res = FALSE;

**while** ((r!=**NULL**)&&(!Res))

**if** ((\*r).Id==q) Res = TRUE; **else** r = (\*r).Next;

**if** (!Res)

{ **//Если дуга отсутствует, то поместим её в граф.**

t = **new** (Trailer); (\*t).Id = q; (\*t).Next = (\*p).Trail;

(\*p).Trail = t; (\*q).Count++;

}

}

**void** Spisok::DeleteGraph (**int** x,**int** y)

**//Функция возвращает указатель Head на структуру**

**//Вирта, соответствующую ориентированному графу**

**//и полученную удалением дуги (x,y).**

{

Lref p,q; **//Рабочие указатели.**

Tref t,r; **//Рабочие указатели.**

Boolean Res; **//Флаг наличия в графе данной дуги.**

**//Определим, существует ли в графе дуга (x,y)?**

p=Search (x); q= Search (y);

**if** ((p!=**NULL**)&&(q!=**NULL**))

{ **//Вершины x и y в графе есть.**

r = (\*p).Trail; Res = FALSE;

**while** ((r!=**NULL**)&&(!Res))

**if** ((\*r).Id==q) Res = TRUE;

**else** r = (\*r).Next;

**if** (Res) **//Если дуга существует, то удалим её.**

**if** (r==(\*p).Trail)

{ (\*p).Trail = (\*(\*p).Trail).Next; **delete** r; (\*q).Count--; }

**else**

{

t = (\*p).Trail;

**while** ((\*t).Next!=r) t = (\*t).Next;

(\*t).Next = (\*(\*t).Next).Next; **delete** r; (\*q).Count--; }

}

}

**void** Spisok::PrintGraph ()

**//Вывод структуры Вирта, заданной указателем**

**//Head и соответствующей ориентированному графу.**

{

Lref p; **//Рабочий указатель.**

Tref q; **//Рабочий указатель.**

p = Head;

**while** (p!=Tail)

{

cout<<"("<<(\*p).Key; q = (\*p).Trail;

**while** (q!=**NULL**)

{ cout<<(\*(\*q).Id).Key; q = (\*q).Next; }

cout<<")"; p = (\*p).Next; cout<<" ";

}

}

**void** Spisok::DeleteY (**int** y)

**//Функция возвращает указатель Head на структуру Вирта,**

**//соответствующую графу с удаленной вершиной y.**

{

Lref p,q; **//Рабочие указатели.**

Tref r,s; **//Рабочие указатели.**

**int** x; **//Рабочая переменная.**

**//Удаление всех дуг (x,y), оканчивающихся в вершине y.**

p = Head;

**while** (p!=Tail)

{ x = (\*p).Key; DeleteGraph (x,y); p = (\*p).Next; }

**//Удаление списка смежности вершины y.**

p=SearchGraph (y); r = (\*p).Trail;

**while** (r!=**NULL**)

{ s = r; r = (\*r).Next; (\*(\*s).Id).Count++; **delete** s; }

**//Удаление узла, содержащего вершину y, из списка заголовочных узлов.**

q = Head;

**if** (q==p) { Head = (\*Head).Next; **delete** q; }

**else** {

**while** ((\*q).Next!=p) q = (\*q).Next;

(\*q).Next = (\*p).Next; **delete** p; }

}

**void** Spisok::Free1Graph (Lref \*Head, Lref \*Tail)

**//Очистка динамической памяти, занятой линейным списком,**

**// заданным указателем \*Head.**

{

**if** (\*Head!=\*Tail)

{ Free1Graph (&(\*\*Head).Next,Tail); **delete** \*Head; \*Head = **NULL**; }

}

**void** Spisok::Free2Graph (Tref \*X)

**//Очистка динамической памяти, занятой линейным списком,**

**// заданным указателем \*X.**

{

**if** (\*X!=**NULL**)

{ Free2Graph (&(\*\*X).Next); **delete** \*X; \*X = **NULL**; }

}

Текст этой программы можно взять [здесь](http://khpi-iip.mipk.kharkiv.edu/library/datastr/book_sod/kgsu/din85_1.zip).

На следующем шаге мы познакомимся с ***модифицированной структурой Вирта***.

## Модифицированные структуры Вирта

На этом шаге мы рассмотрим ***модифицированные структуры Вирта***.

Часто при реализации алгоритмов на ***ориентированных графах*** (см. [1, с.129,143,149]) требуется по заданной конечной вершине некоторой ***дуги ориентированного графа*** восстановить ***список "предшествующих" вершин***. Для успешного решения поставленной задачи модифицируем ранее написанные функции [**MakeGraph()**](http://khpi-iip.mipk.kharkiv.edu/library/datastr/book_sod/kgsu/din_0084.html#MakeGraph), [**AddGraph()**](http://khpi-iip.mipk.kharkiv.edu/library/datastr/book_sod/kgsu/din_0084.html#AddGraph) и [**PrintGraph()**](http://khpi-iip.mipk.kharkiv.edu/library/datastr/book_sod/kgsu/din_0084.html#PrintGraph) с учетом следующего описания типов:

**typedef** **struct** L \*Lref; **//Тип: указатель на заголовочный узел.**

**typedef** **struct** T \*Tref; **//Тип: указатель на дуговой узел.**

**//Описание типа заголовочного узла.**

**typedef** **struct** L

{

**int** Key; **//Имя заголовочного узла.**

**int** Count; **//Количество предшественников.**

**int** Count1; **//Количество последующих вершин.**

Tref Pred; **//Указатель на список смежности, содержащий "предшественников".**

Tref Trail; **//Указатель на список смежности, содержащий "последователей".**

Lref Next; **//Указатель на следующий узел в списке заголовочных узлов.**

} Leader;

**//Описание типа дугового узла.**

**typedef** **struct** T

{

Lref Id;

Tref Next;

} Trailer;

Как и ранее, вершины графа помещаются в связанный список ***заголовочных узлов***, каждый из которых теперь будет содержать ***шесть полей***.

Если указатель **P** указывает на заголовочный узел, представляющий вершину **Q** графа, то

* поле **(\*P).Key** содержит информацию, связанную с этой вершиной (***идентификатор вершины графа***);
* поле **(\*P).Count** содержит количество вершин, "предшествующих" данной;
* поле **(\*P).Count1** содержит количество вершин, "следующих за" данной;
* поле **(\*P).Next** содержит указатель на заголовочный узел, представляющий следующую вершину графа (если такая вершина есть) в списке заголовочных узлов;
* каждый заголовочный узел является заглавным звеном двух списков узлов второго типа, называемых ***дуговыми узлами***. Такие списки мы, как и ранее, будем называть ***списками смежности***.

Каждый узел ***списков смежности*** представляет дугу графа. Поле **(\*P).Trail** указывает на список смежности, представляющий дуги, ***выходящие*** из вершины **Q** графа.

***Второй список смежности*** представляет собой список узлов, представляющих вершины-"предшественники" данной вершины графа. Поле **(\*P).Pred** содержит указатель на этот список.

Каждый узел списков смежности содержит два поля: **Id** и **Next**, причем для списка смежности, заданного указателем **(\*P).Trail**, справедливо следующее: если **Q** указывает на списочный узел, представляющий дугу **(A,B)**, то

* поле **(\*Q).Id** указывает на заголовочный узел, представляющий вершину **B** графа;
* поле **(\*Q).Next** указывает на списочный узел, представляющий следующую дугу, выходящую из вершины **A**графа (если такая дуга есть).

На рисунке показан граф и его ***"улучшенное"*** представление Вирта:

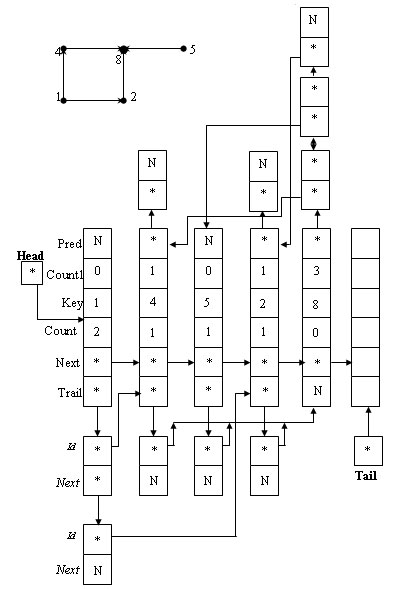


Рис.1. Граф и его улучшенное представление Вирта

Пример. Построение модифицированной структуры Вирта, соответствующей ориентированному графу, вывод ее на экран и добавление ребер.

**#include** <iostream.h>

**#define** TRUE 1

**#define** FALSE 0

**typedef** **int** Boolean;

**typedef** **struct** L \*Lref; **//Тип: указатель на заголовочный узел.**

**typedef** **struct** T \*Tref; **//Тип: указатель на дуговой узел.**

**//Описание типа заголовочного узла.**

**typedef** **struct** L

{

**int** Key; **//Имя заголовочного узла.**

**int** Count; **//Количество предшественников.**

**int** Count1; **//Количество последующих вершин.**

Tref Pred; **//Указатель на список смежности, содержащий "предшественников".**

Tref Trail; **//Указатель на список смежности, содержащий "последователей".**

Lref Next; **//Указатель на следующий узел в**

**//списке заголовочных узлов.**

} Leader;

**//Описание типа дугового узла.**

**typedef** **struct** T

{

Lref Id;

Tref Next;

} Trailer;

**class** Spisok {

**private**:

Lref Head; **//Указатель на список заголовочных узлов.**

Lref Tail; **//Указатель на фиктивный элемент**

**//в конце списка заголовочных узлов.**

Lref SearchGraph(**int**);

Lref Search(**int**);

**public**:

Spisok () {**//Инициализация списка заголовочных узлов.**

Head = Tail = **new** (Leader); }

~Spisok(); **//Деструктор.**

**void** MakeGraph();

**void** AddGraph(**int**, **int**);

**void** DeleteGraph(**int**, **int**);

**void** PrintGraph();

**void** DeleteY(**int**);

**void** Free1Graph(Lref \*, Lref \*);

**void** Free2Graph(Tref \*);

};

**void** main ()

{

Spisok A;

**int** x,y; **//Начало и конец дуги.**

**//Построение и вывод структуры Вирта.**

A.MakeGraph ();

A.PrintGraph (); cout<<endl;

cout<< "Добавим к графу новую дугу...\n";

cout<< "Введите начало дуги: "; cin>>x;

cout<< "Введите конец дуги: "; cin>>y;

A.AddGraph (x,y);

A.PrintGraph (); cout<<endl;

cout<< "Добавим к графу новое ребро...\n";

cout<< "Введите первую концевую вершину ребра: "; cin>>x;

cout<< "Введите вторую концевую вершину ребра: "; cin>>y;

A.AddGraph (x,y);

A.AddGraph (y,x);

A.PrintGraph (); cout<<endl;

cout<< "Удалим из графа заданную дугу...\n";

cout<< "Введите начало дуги: "; cin>>x;

cout<< "Введите конец дуги: "; cin>>y;

A.DeleteGraph (x,y);

A.PrintGraph (); cout<<endl;

cout<< "Удалим из графа заданное ребро...\n";

cout<< "Введите первую концевую вершину ребра: "; cin>>x;

cout<< "Введите вторую концевую вершину ребра: "; cin>>y;

A.DeleteGraph (x,y);

A.DeleteGraph (y,x);

A.PrintGraph (); cout<<endl;

cout<< "Введите удаляемую вершину: "; cin>>y;

A.DeleteY (y);

A.PrintGraph (); cout<<endl;

}

Spisok::~Spisok()

{

**//Очистка структуры Вирта.**

Lref t = Head;

**while** (t!=Tail)

{ Free2Graph (&(\*t).Trail); t = (\*t).Next; }

Free1Graph (&Head,&Tail);

**delete** Tail;

}

Lref Spisok::SearchGraph (**int** w)

**//Функция возвращает указатель на заголовочный узел**

**//с ключом w. Если заголовочный узел отсутствует, то он**

**//добавляется в список. Head - указатель на структуру Вирта.**

{

Lref h;

h = Head; (\*Tail).Key = w;

**while** ((\*h).Key!=w) h = (\*h).Next;

**if** (h==Tail)

**//В списке заголовочных узлов нет узла с ключом w.**

**//Поместим его в конец списка Head.**

{ Tail = **new** (Leader); (\*h).Count = (\*h).Count1 = 0;

(\*h).Trail = (\*h).Pred = **NULL**; (\*h).Next = Tail; }

**return** h;

}

Lref Spisok::Search (**int** w)

**//Функция возвращает указатель на заголовочный узел**

**//ключом w. Если узел отсутствует, то функция возвращает NULL .**

{

Lref h = Head;

(\*Tail).Key = w; **//Поиск "с барьером".**

**while** ((\*h).Key!=w) h = (\*h).Next;

**if** (h==Tail) **//В списке заголовочных узлов нет узла с ключом w.**

h = **NULL**;

**return** h;

}

**void** Spisok::MakeGraph ()

**//Функция возвращает указатель Head на модифицированную структуру**

**//Вирта, соответствующую ориентированному графу.**

{

**int** x,y;

Lref p,q; **//Рабочие указатели.**

Tref t,r; **// Рабочие указатели.**

Boolean Res; **//Флаг наличия дуги.**

cout<<"Вводите начальную вершину дуги: ";

cin>>x;

**while** (x!=0)

{

cout<<"Вводите конечную вершину дуги: "; cin>>y;

**//Определим, существует ли в графе дуга (x,y)?**

p=SearchGraph (x); q=SearchGraph (y);

r = (\*p).Trail; Res = FALSE;

**while** ((r!=**NULL**)&&(!Res))

**if** ((\*r).Id==q) Res = TRUE;

**else** r = (\*r).Next;

**if** (!Res) **//Если дуга отсутствует, то поместим её в граф.**

{ t = **new** (Trailer); (\*t).Id = q;

(\*t).Next = (\*p).Trail; (\*p).Trail = t; (\*q).Count++;

t = **new** (Trailer); (\*t).Id = p;

(\*t).Next = (\*q).Pred; (\*q).Pred = t; (\*p).Count1++; }

cout<<"Вводите начальную вершину дуги: "; cin>>x;

}

}

**void** Spisok::AddGraph (**int** x,**int** y)

**//Добавление дуги (x,y) (если ее не было!) к модифицированной структуре**

**//Вирта, соответствующей ориентированному графу Head.**

{

Lref p,q; **//Рабочие указатели.**

Tref t,r; **//Рабочие указатели.**

Boolean Res; **//Флаг наличия в графе данной дуги.**

**//Определим, существует ли в графе дуга (x,y)?**

p=SearchGraph (x); q=SearchGraph (y);

r = (\*p).Trail; Res = FALSE;

**while** ((r!=**NULL**)&&(!Res))

**if** ((\*r).Id==q) Res = TRUE; **else** r = (\*r).Next;

**if** (!Res)

{ **//Если дуга отсутствует, то поместим её в граф.**

t = **new** (Trailer); (\*t).Id = q; (\*t).Next = (\*p).Trail;

(\*p).Trail = t; (\*q).Count++;

t = **new** (Trailer); (\*t).Id = p;

(\*t).Next = (\*q).Pred; (\*q).Pred = t; (\*p).Count1++;

}

}

**void** Spisok::DeleteGraph (**int** x,**int** y)

**//Функция возвращает указатель Head на структуру**

**//Вирта, соответствующую ориентированному графу**

**//и полученную удалением дуги (x,y).**

{

Lref p,q; **//Рабочие указатели.**

Tref t,r; **//Рабочие указатели.**

Boolean Res; **//Флаг наличия в графе данной дуги.**

**//Определим, существует ли в графе дуга (x,y)?**

p=Search (x); q= Search (y);

**if** ((p!=**NULL**)&&(q!=**NULL**))

{ **//Вершины x и y в графе есть.**

r = (\*p).Trail; Res = FALSE;

**while** ((r!=**NULL**)&&(!Res))

**if** ((\*r).Id==q) Res = TRUE;

**else** r = (\*r).Next;

**if** (Res) **//Если дуга существует, то удалим её.**

**if** (r==(\*p).Trail)

{ (\*p).Trail = (\*(\*p).Trail).Next; **delete** r; (\*q).Count--; }

**else**

{

t = (\*p).Trail;

**while** ((\*t).Next!=r) t = (\*t).Next;

(\*t).Next = (\*(\*t).Next).Next; **delete** r; (\*q).Count--; }

}

}

**void** Spisok::PrintGraph ()

**//Вывод модифицированной структуры Вирта, заданной указателем**

**//Head и соответствующей ориентированному графу.**

{

Lref p; **//Рабочий указатель.**

Tref q; **//Рабочий указатель.**

p = Head;

**while** (p!=Tail)

{

cout<<(\*p).Key<<"(("; q = (\*p).Trail;

**while** (q!=**NULL**)

{ cout<<(\*(\*q).Id).Key<<" "; q = (\*q).Next; }

cout<<")(";

q = p->Pred;

**while** (q!=**NULL**)

{ cout <<(\*(\*q).Id).Key<<" "; q = q->Next; }

cout<<"))";

p = (\*p).Next; cout<<" ";

}

}

**void** Spisok::DeleteY (**int** y)

**//Функция возвращает указатель Head на структуру Вирта,**

**//соответствующую графу с удаленной вершиной y.**

{

Lref p,q; **//Рабочие указатели.**

Tref r,s; **//Рабочие указатели.**

**int** x; **//Рабочая переменная.**

**//Удаление всех дуг (x,y), оканчивающихся в вершине y.**

p = Head;

**while** (p!=Tail)

{ x = (\*p).Key; DeleteGraph (x,y); p = (\*p).Next; }

**//Удаление списка смежности вершины y.**

p=SearchGraph (y); r = (\*p).Trail;

**while** (r!=**NULL**)

{ s = r; r = (\*r).Next; (\*(\*s).Id).Count++; **delete** s; }

**//Удаление узла, содержащего вершину y, из списка заголовочных узлов.**

q = Head;

**if** (q==p) { Head = (\*Head).Next; **delete** q; }

**else** {

**while** ((\*q).Next!=p) q = (\*q).Next;

(\*q).Next = (\*p).Next; **delete** p; }

}

**void** Spisok::Free1Graph (Lref \*Head, Lref \*Tail)

**//Очистка динамической памяти, занятой линейным списком,**

**// заданным указателем \*Head.**

{

**if** (\*Head!=\*Tail)

{ Free1Graph (&(\*\*Head).Next,Tail); **delete** \*Head; \*Head = **NULL**; }

}

**void** Spisok::Free2Graph (Tref \*X)

**//Очистка динамической памяти, занятой линейным списком,**

**// заданным указателем \*X.**

{

**if** (\*X!=**NULL**)

{ Free2Graph (&(\*\*X).Next); **delete** \*X; \*X = **NULL**; }

}

Текст этой программы можно взять [здесь](http://khpi-iip.mipk.kharkiv.edu/library/datastr/book_sod/kgsu/din86_1.zip).

***Замечание****. Вы должны представлять* ***важное различие*** *между представлениями графа в виде матрицы смежностей и представлениями в виде структур смежности [2, с.408].*

*Если граф представлен с помощью* ***матрицы смежностей****, то у нас появляется возможность очевидного прохождения по* ***строке*** *или* ***столбцу*** *матрицы. Просмотр строки эквивалентен нахождению всех дуг, выходящих из данной вершины, что может быть эффективно реализовано при связанном представлении путем прохождения списка всех дуговых узлов, начинающегося с данного заголовочного узла.*

*Однако просмотр* ***столбца*** *матрицы смежностей эквивалентен нахождению всех дуг, оканчивающихся в данном узле, а для этой процедуры нет соответствующего простого метода для связанного представления, если не воспользоваться* ***модифицированной структурой Вирта****.*

*Конечно, Вы должны выбрать из этих представлений наиболее подходящее, исходя из требований конкретной задачи с учетом эффективности использования памяти и минимизации времени исполнения.*

(1) Липский В. Комбинаторика для программистов: Пер. с польск. - М.: Мир, 1988. - 213 с.

(2) Лэнгсам Й., Огенстейн М., Тененбаум А. Структуры данных для персональных ЭВМ: Пер. с англ. - М.: Мир, 1989. - 568 с.

На следующем шаге мы познакомимся с ***представлением отношений***.

## Представление отношений

На этом шаге мы определим ***различные виды отношений***.

***Определение.***

Под ***бинарным отношением на множестве* M** мы понимаем произвольное подмножество **E** множества **MxM**.

Рассмотрим ориентированный граф, в котором любая дуга **(v,w)** имеется только в том случае, если элементы **v** и **w**, представляемые вершинами **v** и **w**, находятся в ***данном бинарном отношении* r**, т.е. **vrw**. Такой граф является исчерпывающей и наглядной формой представления отношения **r**, так как он полностью перечисляет все упорядоченные пары вершин-элементов, для которых отношение **r** имеет место.

Графы отношений могут обладать специальными свойствами: рефлексивностью, симметричностью, антисимметричностью, транзитивностью и т.д., отражающими соответствующие свойства отношений.

***Определение 1.***

Отношение **r** называется ***рефлексивным*** на множестве **M**, если для всякого **a** принадлежащего **M** верно **ara**. Отношение **r** называется ***нерефлексивным*** на множестве **M**, если ни для какого **a** принадлежащего **M** не выполняется **ara**.

Будем говорить, что граф является ***рефлексивным***, если каждая вершина имеет петлю, и ***антирефлексивным***, если ни одна вершина петли не имеет.

***Определение 2.***

Отношение **r** называется ***симметричным*** на множестве **M**, если для каждой пары **a** и **b** элементов **M** из **arb**следует **bra**. Отношение **r** называется ***антисимметричным*** на множестве **M**, если для несовпадающих элементов **a** и **b** из **arb** следует не **bra**.

Будем говорить, что граф является ***симметричным***, если каждой дуге **(v,w)** соответствует дуга **(w,v)**, и ***антисимметричным***, если каждая дуга **(v,w)** исключает существование дуги **(w,v)** (заметим, что антисимметричный граф может как иметь петли, так и не иметь их!).

***Определение 3.***

Отношение **r** называется ***транзитивным*** на множестве **M**, если для любых трех элементов **a, b** и **g**, принадлежащих **M**, из **arb** и **brg** следует **arg**. Отношение **r** называется ***антитранзитивным*** на множестве **M**, если для любых трех элементов **a, b** и **g**, принадлежащих **M**, из **arb** и **brg** следует не **arg**.

Будем говорить, что граф является ***транзитивным***, если существование дуг **(v,w)** и **(w,u)** означает существование дуги **(v,u)**, и ***антитранзитивным***, если существование дуг **(v,w)** и **(w,u)** означает несуществование указанной дуги.

***Определение 4.***

Отношение **r** на множестве **M**, которое одновременно рефлексивно, симметрично и транзитивно, называется ***отношением эквивалентности***.

***Графом отношения эквивалентности*** называется граф, являющийся

* ***рефлексивным***,
* ***симметричным*** и
* ***транзитивным***.

***Определение 5.***

Отношение **r** называется ***полным отношением***, если в нем для всякой пары **a, b** несовпадающих элементов множества **M** имеет место **arb**, либо **bra**. Отношение, не являющееся полным, называется ***неполным***.

***Граф полного отношения*** (***полный граф***) характеризуется наличием хотя бы одной дуги для любой пары вершин. В графе неполного отношения некоторые пары вершин не соединены дугами.

***Определение 6.***

Отношение **r** называется ***отношением порядка***, если оно

* ***антисимметричное*** и
* ***транзитивное***.

Соответственно ***графом отношения порядка*** называется антисимметричный и транзитивный граф.

***Определение 7.***

Отношение **r** называется ***отношением полного порядка*** (***полным порядком***), если оно

* ***антисимметричное***,
* ***транзитивное*** и
* ***полное***.

Соответственно ***графом отношения полного порядка*** называется антисимметричный, транзитивный и полный граф.

Отношение **r** называется ***отношением неполного порядка*** (***неполным порядком***), если оно

* ***антисимметричное***,
* ***транзитивное*** и
* ***неполное***.

Соответственно ***графом отношения неполного порядка*** называется антисимметричный, транзитивный и неполный граф.

***Определение 8.***

Отношение называется ***отношением строгого порядка***, если оно

* ***антирефлексивно***,
* ***антисимметрично*** и
* ***транзитивно***.

Соответственно ***графом отношения строгого порядка*** называется антирефлексивный, антисимметричный и транзитивный граф.

Отношение называется ***отношением нестрогого порядка***, если оно

* ***рефлексивно***,
* ***антисимметрично*** и
* ***транзитивно***.

Соответственно ***графом отношения нестрогого порядка*** называется рефлексивный, антисимметричный и транзитивный граф.

Конечно, отношение строгого порядка может быть как полным, так и неполным, и отношение нестрогого порядка тоже может быть как полным, так и неполным.

Определения разных видов порядка сведем в таблицу [1, с.87]:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Таблица 1. **Определения разных видов порядка** | | | | | | |
| **Порядки** | **Отношения** | | | | | |
| **Антисимметричное** | **Транзитивное** | **Рефлексивное** | **Антирефлексивное** | **Полное** | **Неполное** |
| Строгий | + | + |  | + |  |  |
| Нестрогий | + | + | + |  |  |  |
| Полный | + | + |  |  | + |  |
| Неполный | + | + |  |  |  | + |

(1) Березина Л.Ю. Графы и их применение. - М.: Просвещение, 1979. - 143 с.

На следующем шаге мы познакомимся с ***топологической сортировкой***.

## Топологическая сортировка

На этом шаге мы рассмотрим ***общие принципы организации топологической сортировки***.

***Определение [1, с.324].***

Множество **M** называется ***частично упорядоченным***, если над его элементами определено отношение, которое мы назовем ***"x предшествует y"*** и обозначим **x<<y**, удовлетворяющее следующим свойствам для любых элементов **x, y** и **z** из **M**:

* не **x<<x** (***антирефлексивность***),
* если **x<<y**, то не **y<<x** (***антисимметричность***),
* если **x<<y** и **y<<z**, то **x<<z** (***транзитивность***).

Мы будем предполагать, что **M** - ***конечное множество***.

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

***Проблема топологической сортировки*** состоит в том, чтобы "перевести частичное упорядочение в линейное упорядочение", т.е. расположить элементы в такую последовательность **a1,a2, ..., an**, что если **aj<<ak**, то **j<k** [1, с.325]. Существование такого расположения не является непосредственно очевидным, однако ***оно заведомо невозможно***, если имеется хотя бы один контур.

Как найти одно из возможных линейных упорядочений? Рецепт достаточно прост. Мы начинаем с того, что выбираем ***какой-либо элемент***, которому не предшествует никакой другой (хотя бы один такой элемент существует, иначе имелся бы цикл). Этот элемент помещается в начало списка и исключается из множества **M**. Оставшееся множество по-прежнему частично упорядочено; таким образом, можно вновь применить тот же самый алгоритм, пока множество не станет пустым. Этот алгоритм оказался бы непригодным в единственном случае, когда образовалось бы непустое частично упорядоченное множество, в котором каждому элементу предшествует другой.

Для того, чтобы подробнее сформулировать этот алгоритм, нужно ***описать структуры данных***, а также выбрать представление **M** и отношения порядка. Это представление зависит от выполняемых действий, особенно от ***операции выбора элемента без предшественников***. Поэтому каждый элемент удобно представить тремя характеристиками:

* ***ключом***,
* ***множеством*** следующих за ним элементов ("последователей") и
* ***счетчиком*** предшествующих элементов ("предшественников").

Поскольку **n** - число элементов в **M** не задано априори, то это множество удобно организовать в виде ***линейного однонаправленного списка***. Следовательно, каждый узел содержит еще поле, связывающее его со следующим узлом списка.

Мы будем считать, что ключи - это целые числа (необязательно последовательные от 1 до **n**). Аналогично множество последователей каждого элемента можно представить в виде ***линейного однонаправленного списка***. Каждый узел списка последователей неким образом идентифицирован и связан со следующим узлом этого списка.

Если мы назовем узлы главного списка, в котором каждый элемент из **M** содержится ровно один раз, ***ведущими*(Leaders)**, а узлы списка последователей ***ведомыми* (Trailers)**, то мы получим такие описания типов данных [2]:

**struct** Leader

{

**int** Key;

**int** Count;

Trailer\* Trail;

Leader\* Next;

};

**struct** Trailer

{

Leader\* Id;

Trailer\* Next;

};

Теперь легко видеть, что описанная структура является ***структурой Вирта*** некоторого ориентированного графа.

Первая часть программы топологической сортировки должна преобразовать входные данные в структуру списка. Это производится последовательным чтением пар ключей **x** и **y** (**x<<y**). Мы предполагаем, что последовательность входных пар ключей заканчивается дополнительным нулем.

Обозначим ссылки на их представления в списке ведущих через **p** и **q**. Эти записи ищутся в списке и, если их там нет, добавляются к нему. Эту задачу выполняет функция **L ("Located")**. Затем к списку ведомых для элемента **x**добавляется новый узел, идентифицированный как **y**, счетчик предшественников для **y** увеличивается на 1. Такой алгоритм соответствует ***фазе ввода***.

**//Фаза ввода.**

cout << "Задайте отношение частичного поpядка...\n";

cout << "Элемент ";

cin >> x;

cout << " пpедшествует элементу ";

**while** (x!=0)

{

cin >> y;

p = L (x);

q = L (y);

t = **new** (Trailer); t->Id = q; t->Next = p->Trail;

p->Trail = t; q->Count += 1;

cout << "Элемент ";

cin >> x;

cout << " пpедшествует элементу ";

}

В этом фрагменте программы есть обращения к функции **L(w)**, возвращающей ссылку на компоненту списка с ключом **w**.

На рисунке показана структура, сформированная при обработке входных данных вида: **1<<4, 1<<2, 4<<8, 5<<8, 2<<8** с помощью этого алгоритма:

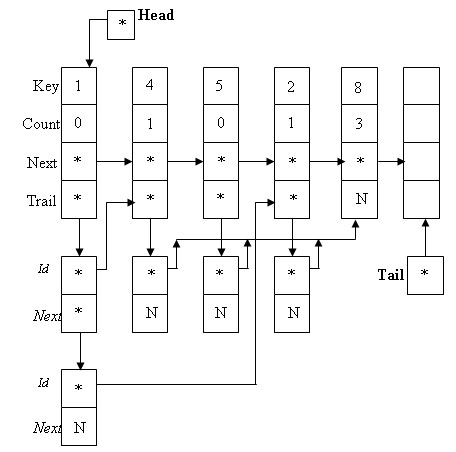


Рис.1. Структура Вирта

После того, как на фазе ввода построена некоторая структура данных, можно провести саму топологическую сортировку, описанную выше. Но поскольку она состоит в последовательном выборе элемента с нулевым счетчиком предшественников, видимо, разумно вначале собрать все такие элементы в некоторый новый список. Поскольку мы знаем, что исходный список ведущих впоследствии не понадобится, то же самое поле **Next** можно использовать повторно для помещения в список ведущих, не имеющих предшественников. ***Такая замена одного списка на другой***часто встречается при работе ***со списками***.

Это подробно описано в следующем программном фрагменте:

**//Поиск ведущих с нулевым количеством предшественников.**

p = Head; Head = **NULL**;

**while** (p!=Tail)

{

q = p; p = p->Next;

**if** (q->Count==0)

{ q->Next = Head; Head = q; }

}

Для удобства новая цепочка строится ***в обратном порядке***.

Для нашего примера мы увидим, что список ведущих заменяется на список вида:

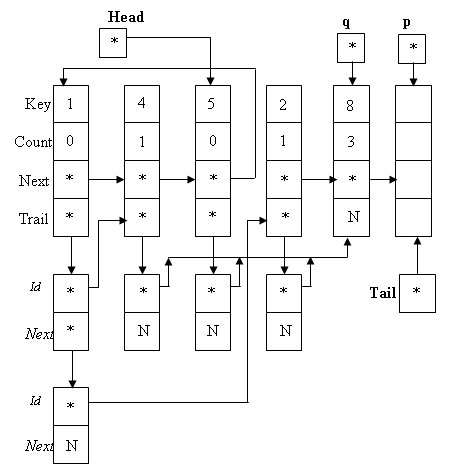


Рис.2. Результат преобразования

После всех этих подготовительных действий, направленных на то, чтобы выработать подходящее представление частично упорядоченного множества **M**, мы можем, наконец, перейти к собственно топологической сортировке, т.е. формированию выходной последовательности.

Это можно описать следующим образом:

**//Фаза вывода.**

cout << "Результат...\n";

q = Head;

**while** (q!=NULL)

{

**//Вывести элемент, а затем исключить его.**

cout << q->Key << " "; z -= 1; t = q->Trail; q = q->Next;

**while** (t!=NULL)

{

**// Уменьшить счетчик предшественников у всех его**

**// последователей в списке ведомых t; если какой-**

**// либо счетчик стал равен 0, то добавить этот**

**// элемент к списку ведущих q;**

**// p - вспомогательная переменная, указывающая на**

**// ведущий узел, счетчик которого нужно уменьшить**

**// и проверить на равенство нулю.**

p = t->Id; p->Count -= 1;

**if** (p->Count==0) **//Включение \*p в список ведущих.**

{ p->Next = q; q = p; }

t = t->Next;

}

}

На этом разработка программы топологической сортировки завершается. Обратите внимание, что был введен счетчик z для подсчета ведущих узлов, сформированных на фазе ввода. Этот счетчик уменьшается каждый раз, когда ведущий узел выводится на фазе вывода. Поэтому он должен вновь стать равным нулю в конце работы программы. Если это не так, то в структуре остались элементы и среди них нет таких, у которых отсутствуют предшественники. Очевидно, что в этом случае множество **M** не является частично ***упорядоченным***.

Приведенная выше программа фазы вывода служит примером работы со списком, который ***"пульсирует"***, т.е. элементы которого добавляются и удаляются в непредсказуемом порядке. Следовательно, это пример процесса, полностью использующего ***гибкость***, которую обеспечивает связанный список.

(1) Кнут Д. Искусство программирования для ЭВМ. Т.1: Основные алгоритмы. - M.: Мир, 1976. - 736 с.

(2) Вирт H. Алгоритмы + структуры данных = программы. - М.: Мир, 1985. - 406 с.

Со следующего шага мы начнем приводить ***примеры программ с использованием топологической сортировки***.

## Первый пример использования топологической сортировки

На этом шаге мы начнем рассматривать ***примеры использования топологической сортировки***.

Пример 1 [2, с.211-219].

**#include** <iostream.h>

**typedef** **struct** Leader \*Lref; **//Тип: указатель на заголовочный узел.**

**typedef** **struct** Trailer \*Tref; **//Тип: указатель на дуговой узел.**

**//Описание типа заголовочного узла.**

**typedef** **struct** Leader

{

**int** Key; **//Информационное поле.**

**int** Count; **//Количество предшественников.**

Tref Trail;

Lref Next;

};

**//Описание типа дугового узла.**

**typedef** **struct** Trailer

{

Lref Id;

Tref Next;

};

**class** Spisok {

**private**:

Lref Head; **//Указатель на список заголовочных узлов.**

Lref Tail; **//Указатель на фиктивный элемент**

**//в конце списка заголовочных узлов.**

**int** z; **//Количество узлов, не имеющих предшественников.**

**public**:

Spisok () {**//Инициализация списка заголовочных узлов.**

Head = Tail = **new** (Leader); z = 0;};

Lref L (**int**);

**void** Poisk();

**void** Vyvod();

};

**void** Spisok::Poisk()

{

Lref p,q; **// Рабочие указатели.**

p = Head; Head = **NULL**;

**while** (p!=Tail)

{

q = p; p = p->Next;

**if** (q->Count==0)

{ q->Next = Head; Head = q; }

}

}

Lref Spisok::L (**int** w)

**//Функция возвращает указатель на ведущего с ключом w.**

{

Lref h = Head;

Tail->Key = w;

**while** (h->Key!=w) h = h->Next;

**if** (h==Tail)

**// В списке нет элемента с ключом w.**

{

Tail = **new** (Leader); z++;

h->Count = 0; h->Trail = **NULL**; h->Next = Tail;

}

**return** h;

}

**void** Spisok::Vyvod()

{

Lref p,q; **// Рабочие указатели.**

Tref t;

cout << endl;

cout << "Результат...\n";

q = Head;

**while** ( q!=NULL)

{

cout << q->Key << " ";

z--;

t = q->Trail; q = q->Next;

**while** (t!=NULL)

{

p = t->Id; p->Count--;

**if** (p->Count==0) **// Включение (\*p) в список ведущих.**

{ p->Next = q; q = p; }

t = t->Next;

}

}

**if** (z!=0)

cout << "\nМножество не является частично упорядоченным!\n";

}

**void** main()

{

Spisok A;

Lref p,q; **// Рабочие указатели.**

Tref t;

**int** x,y; **// Рабочие переменные.**

**// Фаза ввода.**

cout << "Задайте отношение частичного порядка...\n";

cout << "Элемент ";

cin >> x;

cout << " предшествует элементу ";

**while** (x!=0)

{

cin >> y;

p = A.L(x); q = A.L(y);

t = **new** (Trailer); t->Id = q; t->Next = p->Trail;

p->Trail = t; q->Count += 1;

cout << "Элемент ";

cin >> x;

cout << " предшествует элементу ";

}

**// Поиск ведущих с нулевым количеством предшественников.**

A.Poisk();

**// Фаза вывода.**

A.Vyvod();

}

Текст этой программы можно взять [здесь](http://khpi-iip.mipk.kharkiv.edu/library/datastr/book_sod/kgsu/din89_1.zip).

Тестовые пpимеpы:

1) ***данный тестовый пример заимствован у Д.Кнута*** [1, с.324-325].

Система отношений порядка имеет вид:

9>2, 3>7, 7>5, 5>8, 8>6, 4>6, 1>3, 7>4, 9>5, 2>8.

Программа топологической сортировки выдаст следующий результат:

1 3 7 4 9 2 5 8 6

2) ***Приведем более содержательный пример использования программы топологической сортировки*** [3, с.261-262].

Разнообразие вин, отражающих почтенную традицию и свидетельствующее об исключительном богатстве палитры французских виноделов, кажется, представляет бесконечные возможности для сервировки хорошего обеда. В действительности же имеются ограничения, выраженные следующими двумя общими правилами:

* не принято подавать за обедом более четырех вин, не считая шампанского;
* последовательность вин на столе подчиняется некоторым соотношениям порядка, признаваемым всеми знатоками. Эти соотношения порядка таковы:
* белое сухое < белое бархатистое,
* белое бархатистое < белое сладкое,
* красное легкое < красное крепкое,
* белое (за исключением сладкого) < красное,
* крепкое < белое сладкое.

При этом знак < указывает, что вино, стоящее слева от него, должно быть подано прежде вина, которое стоит справа. Мы хотим отметить то, что эти соотношения вносят в любой винный погреб частичное упорядочение с точки зрения математика.

Закодируем сорта вин следующим образом:

1 - белое сухое,

2 - белое бархатистое,

3 - белое сладкое,

4 - красное легкое,

5 - красное крепкое.

Тогда система отношений порядка примет вид:

1<2, 2<3, 4<5, 1<4, 1<5, 2<4, 2<5, 4<3, 5<3,

а программа топологической сортировки выдаст следующий результат:

1 2 4 5 3

Hо так как не принято подавать за обедом более ***четырех вин***, а шампанское в нашем перечне сортов вин отсутствует, то возможны лишь следующие варианты:

2 4 5 3

1 4 5 3

1 2 5 3

1 2 4 3

1 2 4 5

3) Этот тестовый пример базируется на идее об использовании топологической сортировки, упомянутой в монографиях [2, с.212; 4, с.311].

В учебных программах для высших учебных заведений одни предметы опираются на материал других, поэтому некоторые курсы студенты должны прослушать раньше других. Если курс **v** содержит материал для курса **w**, мы пишем **v<<w**. Топологическая сортировка курсов означает чтение курсов в таком порядке, чтобы ни один курс не читался раньше того, на материале которого он основан.

Поставим задачу: топологически упорядочить курсы по

* ***математике***,
* ***дискретной математике*** и
* ***информатике***.

для студентов педагогических вузов. Наш взгляд на "упорядоченность" курсов достаточно полно отражает следующая таблица:

|  |  |  |
| --- | --- | --- |
| Таблица 1. **Упорядоченность курсов** | | |
| **Номер курса** | **Название курса** | **Последующие курсы** |
| C1 | Дискретная математика | C2,C3,С4,C5,C6 |
| C2 | Теория кодирования | C10 |
| C3 | Теория автоматов | C5,C11 |
| C4 | Машинная арифметика | C10 |
| C5 | Формальные языки и грамматики | C11 |
| C6 | Теория графов | C8 |
| C7 | Введение в программирование | C8,C18 |
| C8 | Структуры данных | C9,С11,C12,С14,C15,C16,C25 |
| C9 | Анализ алгоритмов | C16,C27 |
| C10 | Устройство ЭВМ | C12 |
| C11 | Теория компиляторов | C12,C14 |
| C12 | Операционные системы | Нет |
| C13 | Базы данных | Нет |
| C14 | Парадигмы программирования | С15,C23 |
| C15 | Искусственный интеллект | Нет |
| C16 | Информационный поиск | C13,C15 |
| C17 | Реляционная алгебра и реляционное исчисление | C13 |
| C18 | Доказательство правильности программ | C14 |
| C19 | Математическая логика | C15,С18,C24 |
| C20 | Теория взаимодействующих последовательных процессов | C12,C14 |
| C21 | Алгебра | C8,C17,C20,C23 |
| C22 | Математический анализ | C9,C23 |
| C23 | Вычислительная математика | Нет |
| C24 | Теория алгоритмов | C2,C7,C9,C14 |
| C25 | Машинная графика | C27 |
| C26 | Геометрия | C25 |
| C27 | Вычислительная геометрия | Нет |

Применив программу топологической сортировки, получим последовательность целых чисел:

26 22 21 17 20 19 24 7 18 1 2 3 5 4 10 6 8 9 16 13 11 12 14 23 15 25 27

Заметим, что в приведенной последовательности допустима перестановка чисел 15 и 23. С учетом этого, получим расположение курсов достаточно крупными блоками:

* ***Классическая математика*** (*алгебра, геометрия, математический анализ, математическая логика и теория алгоритмов*): **26 22 21 17 20 19 24**.
* ***Введение в программирование***: **17 18**.
* ***Дискретная математика***: **1 2 3 5 4**.
* ***Архитектура вычислительных систем***: **10**.
* ***Структуры данных и анализ алгоритмов***: **6 8 9**.
* ***Программное обеспечение вычислительных систем***: **16 13 11 12 14 15.**
* ***Вычислительная математика***: **23**.
* ***Алгоритмы машинной графики***: **25 27**.

***Замечание [1, с.332]****. Метод топологической сортировки был впервые опубликован* ***Каном (A.B.Kahn, 1962)****. Сам факт возможности топологической сортировки был доказан в статье* ***Шпильрайна (E.Szpilrajn, 1930)****.*

(1) Кнут Д. Искусство программирования для ЭВМ. Т.1: Основные алгоритмы. - M.: Мир, 1976. - 736 с.

(2) Вирт H. Алгоритмы + структуры данных = программы. - М.: Мир, 1985. - 406 с.

(3) Кофман А., Фор Р. Займемся исследованием операций. - М.: Мир, 1966, с.261-262.

(4) Tucker, Allen B. Computer Science: A Second Course Using Modula-2. McGraw-Hill, Inc. - 401 pp.

На следующем шаге мы продолжим рассматривать ***примеры использования топологической сортировки***.

## Второй пример использования топологической сортировки

На этом шаге мы рассмотрим ***еще пример использования топологической сортировки*** .

Ниже мы приведем реализацию некоторого алгоритма ***детерминированного выбора ведущего элемента*** с нулевым количеством предшественников при топологической сортировке. Установите самостоятельно в чем его идея!

**#include** <iostream.h>

**typedef** **struct** Leader \*Lref; **//Тип: указатель на заголовочный узел.**

**typedef** **struct** Trailer \*Tref; **//Тип: указатель на дуговой узел.**

**//Описание типа заголовочного узла.**

**typedef** **struct** Leader

{

**int** Key; **//Информационное поле.**

**int** Count; **//Количество предшественников.**

**int** CountF; **//Количество последователей.**

Tref Trail;

Lref Next;

};

**//Описание типа дугового узла.**

**typedef** **struct** Trailer

{

Lref Id;

Tref Next;

};

**class** Spisok {

**private**:

Lref Head; **//Указатель на список заголовочных узлов.**

Lref Tail; **//Указатель на фиктивный элемент**

**//в конце списка заголовочных узлов.**

**int** z; **//Количество узлов, не имеющих предшественников.**

**int** Length\_List(Tref);

**void** Sorting (Lref);

**public**:

Spisok () {**//Инициализация списка заголовочных узлов.**

Head = Tail = **new** (Leader); z = 0;};

Lref L (**int**);

**void** Poisk();

**void** Vyvod();

**void** Zapoln();

};

**void** Spisok::Zapoln()

**// Заполнение полей CountF узлов построенного графа.**

{

Lref p; **// Рабочий указатель.**

p = Head;

**while** ( p!=Tail)

{

p->CountF = Length\_List (p->Trail);

p = p->Next;

}

}

**void** Spisok::Poisk()

{

Lref p,q; **// Рабочие указатели.**

p = Head; Head = **NULL**;

**while** (p!=Tail)

{

q = p; p = p->Next;

**if** (q->Count==0)

{ q->Next = Head; Head = q; }

}

}

Lref Spisok::L (**int** w)

**//Функция возвращает указатель на ведущего с ключом w.**

{

Lref h = Head;

Tail->Key = w;

**while** (h->Key!=w) h = h->Next;

**if** (h==Tail)

**// В списке нет элемента с ключом w.**

{

Tail = **new** (Leader); z++;

h->Count = 0; h->Trail = **NULL**; h->Next = Tail;

}

**return** h;

}

**void** Spisok::Vyvod()

{

Lref p,q; **// Рабочие указатели.**

Tref t;

cout << "\nРезультат топологической сортировки:\n";

q = Head;

**while** ( q!=NULL)

{

Sorting (q);

cout << q->Key << " ";

z--;

t = q->Trail; q = q->Next;

**while** (t!=NULL)

{

p = t->Id; p->Count--;

**if** (p->Count==0) **// Включение (\*p) в список ведущих.**

{ p->Next = q; q = p; }

t = t->Next;

}

}

**if** (z!=0)

cout << "\nМножество не является частично упорядоченным!\n";

}

**int** Spisok::Length\_List (Tref T)

**// Функция позволяет найти количество элементов в линейном списке.**

**// T - указатель на звено, следующее за заглавным звеном.**

{

**if** (T==NULL) **return** 0;

**else** **return** 1 + Length\_List(T->Next);

}

**void** Spisok::Sorting (Lref Head)

**// Сортировка списка элементов по полю CountF**

**// в порядке убывания.**

**// 21.10.93 (c) Коврижных Д. & Швецкий М.В.**

{

Lref UkZv\_1,UkZv\_2;

**int** A,B,C;

Tref D;

Lref UkZv\_3; **// Рабочий указатель.**

Tref UkZv\_4; **// Рабочий указатель.**

UkZv\_1 = Head;

**while** ( UkZv\_1!=NULL )

{

UkZv\_2 = UkZv\_1->Next;

**while** (UkZv\_2!=NULL)

{

**if** (UkZv\_1->CountF < UkZv\_2->CountF)

{

A = UkZv\_1->Key;

B = UkZv\_1->Count;

C = UkZv\_1->CountF;

D = UkZv\_1->Trail;

UkZv\_1->Key = UkZv\_2->Key;

UkZv\_1->Count = UkZv\_2->Count;

UkZv\_1->CountF = UkZv\_2->CountF;

UkZv\_1->Trail = UkZv\_2->Trail;

UkZv\_2->Key = A;

UkZv\_2->Count = B;

UkZv\_2->CountF = C;

UkZv\_2->Trail = D;

UkZv\_3 = Head;

**while** (UkZv\_3!=NULL)

{

UkZv\_4 = UkZv\_3->Trail;

**while** (UkZv\_4!=NULL)

{

**if** (UkZv\_4->Id==UkZv\_1)

UkZv\_4->Id = UkZv\_2;

**else**

**if** ( UkZv\_4->Id==UkZv\_2 ) UkZv\_4->Id = UkZv\_1;

UkZv\_4 = UkZv\_4->Next;

}

UkZv\_3 = UkZv\_3->Next;

}

}

UkZv\_2 = UkZv\_2->Next;

}

UkZv\_1 = UkZv\_1->Next;

}

}

**void** main()

{

Spisok A;

Lref p,q; **// Рабочие указатели.**

Tref t;

**int** x,y; **// Рабочие переменные.**

**// Фаза ввода.**

cout << "Задайте отношение частичного порядка...\n";

cout << "Элемент ";

cin >> x;

cout << " предшествует элементу ";

**while** (x!=0)

{

cin >> y;

p = A.L(x); q = A.L(y);

t = **new** (Trailer); t->Id = q; t->Next = p->Trail;

p->Trail = t; q->Count += 1;

cout << "Элемент ";

cin >> x;

cout << " предшествует элементу ";

}

**// Заполнение полей CountF узлов построенного графа.**

A.Zapoln();

**// Поиск ведущих с нулевым количеством предшественников.**

A.Poisk();

**// Фаза вывода.**

A.Vyvod();

}

Текст этой программы можно взять [здесь](http://khpi-iip.mipk.kharkiv.edu/library/datastr/book_sod/kgsu/din90_1.zip).

На следующем шаге мы продолжим рассматривать ***примеры использования топологической сортировки***.

## Третий пример использования топологической сортировки

На этом шаге мы рассмотрим ***еще пример использования топологической сортировки***.

В монографии [1,с.335,упр.23] ***Д.Кнут*** пишет: "Когда алгоритм топологической сортировки заканчивает работу из-за того, что во входной информации обнаружен ***цикл***, то невелика будет польза, если он остановится и сообщит: "Встретился цикл". Полезно в этом случае напечатать один из циклов, показывая ту часть информации, в которой имеется ошибка."

Дополним алгоритм топологической сортировки так, чтобы в нем предусматривался ***вывод какого-нибудь найденного контура на печать***. Воспользуемся алгоритмом, приведенном в [1, с.647].

Для того, чтобы читатель смог разобраться в реализации алгоритма на языке **С++**, мы (редчайший случай!) используем в программе ***метки*** и ***оператор* goto**.

**#include** <iostream.h>

**typedef** **struct** Leader \*Lref; **//Тип: указатель на заголовочный узел.**

**typedef** **struct** Trailer \*Tref; **//Тип: указатель на дуговой узел.**

**#define** FALSE 0

**#define** TRUE 1

**//Описание типа заголовочного узла.**

**struct** Leader

{

**int** Key; **//Информационное поле.**

**int** Count; **//Количество предшественников.**

Tref Trail; **//Указатель на список последователей.**

Lref Next;

};

**//Описание типа дугового узла.**

**struct** Trailer

{

Lref Id;

Tref Next;

};

**class** Spisok {

**private:**

Lref Head; **//Указатель на список заголовочных узлов.**

Lref Tail; **//Указатель на фиктивный элемент**

**//в конце списка заголовочных узлов.**

**int** z; **//Количество узлов, не имеющих предшественников.**

**public:**

Spisok () {**//Инициализация списка заголовочных узлов.**

Head = Tail = **new** (Leader); z = 0;};

Lref L (**int**);

**void** Poisk();

**int** Vyvod();

**void** Neupor();

};

**int** VspomSet[256]; **// Множество, содержащее вершины графа,**

**// имеющие нулевое количество предшественников.**

**// По умолчанию элементы инициалируются 0.**

Lref Spisok::L (**int** w)

**//Функция возвращает указатель на ведущего с ключом w.**

{

Lref h = Head;

Tail->Key = w;

**while** (h->Key!=w) h = h->Next;

**if** (h==Tail)

**// В списке нет элемента с ключом w.**

{

Tail = **new** (Leader); z++;

h->Count = 0; h->Trail = **NULL**; h->Next = Tail;

}

**return** h;

}

**void** Spisok::Poisk()

{

Lref p,q; **// Рабочие указатели.**

p = Head; Head = **NULL**;

**while** (p!=Tail)

{

q = p; p = p->Next;

**if** (q->Count==0)

{ q->Next = Head; Head = q; }

}

}

**int** Spisok::Vyvod()

{

Lref p,q; **// Рабочие указатели.**

Tref t;

cout << "\nРезультат топологической сортировки:\n";

q = Head;

**while** ( q!=NULL)

{

cout << q->Key << " "; VspomSet[q->Key] = 1;

z--;

t = q->Trail; q = q->Next;

**while** (t!=NULL)

{

p = t->Id; p->Count--;

**if** (p->Count==0) **// Включение (\*p) в список ведущих.**

{ p->Next = q; q = p; }

t = t->Next;

}

}

**if** (z!=0)

{

cout << "\nМножество не является частично упорядоченным!\n";

**return** 1;

}

**else** **return** 0;

}

**void** Spisok::Neupor()

{

Lref p,q,r; **// Рабочие указатели.**

Tref t;

**int** Flag;

cout << "Имеется следующий контур (контур печатается ";

cout << "в обратном порядке):\n";

**//В дальнейшем работаем только с копией основного списка.**

p = Head;

**while** (p!=Tail)

{

**if** ( VspomSet[p->Key] ) p->Trail = **NULL**;

p = p->Next;

}

**// -------**

p = Head;

**while** ( p!=Tail )

{ p->Count = 0; p = p->Next; }

**// ---------------------------------------**

p = Head;

**while** ( p!=Tail )

{

r = p; t = p->Trail; p->Trail = **NULL**;

**while** ( t!=NULL )

{

**if** (t!=NULL && t->Id->Count==0) t->Id->Count = r->Key;

t = t->Next;

}

p = p->Next;

}

**// ----------------------**

p = Head; Flag = FALSE;

**while** (p!=Tail && !Flag)

**if** ( p->Count!=0 ) Flag = TRUE;

**else** p = p->Next;

**// ---------------------------------------**

t = **new** (Trailer); **// Создали вспомогательную запись.**

A:

p->Trail = t;

q = Head; Flag = FALSE;

**while** (q!=Tail && !Flag)

**if** ( q->Key==p->Count ) Flag = TRUE;

**else** q = q->Next;

**if** ( q->Trail==NULL ) { p = q; goto A; }

**// -----------------------------------**

z = p->Key; **// Сохранили начало контура.**

B:

cout << p->Key << " ";

p->Trail = **NULL**;

q = Head; Flag = FALSE;

**while** (q!=Tail && !Flag)

**if** ( q->Key==p->Count ) Flag = TRUE;

**else** q = q->Next;

**if** ( q->Trail!=NULL ) { p = q; goto B; }

cout << z; **// Напечатали начало контура.**

}

**void** main()

{

Spisok A,A\_Copy;

Lref p,q; **// Рабочие указатели.**

Tref t;

Lref p1,q1; **// Рабочие указатели для создания копии списка.**

Tref t1;

**int** x,y; **// Рабочие переменные.**

**// Фаза ввода.**

cout << "Задайте отношение частичного порядка...\n";

cout << "Элемент ";

cin >> x;

cout << " предшествует элементу ";

**while** (x!=0)

{

cin >> y;

**//Построение основного списка.**

p = A.L(x); q = A.L(y);

t = **new** (Trailer); t->Id = q; t->Next = p->Trail;

p->Trail = t; q->Count += 1;

**//Построение копии основного списка.**

p1 = A\_Copy.L(x); q1 = A\_Copy.L(y);

t1 = **new** (Trailer); t1->Id = q1; t1->Next = p1->Trail;

p1->Trail = t1; q1->Count += 1;

cout << "Элемент ";

cin >> x;

cout << " предшествует элементу ";

}

**// Поиск ведущих с нулевым количеством предшественников.**

A.Poisk();

**// Фаза вывода.**

**if** (A.Vyvod()) **//Если множество неупорядоченное,**

**//то работаем с копией.**

A\_Copy.Neupor();

}

Текст этой программы можно взять [здесь](http://khpi-iip.mipk.kharkiv.edu/library/datastr/book_sod/kgsu/din91_1.zip).

Тестовые примеры:

* для системы отношений частичного порядка:
* 8>5, 2>3, 2>1, 5>2, 4>1, 3>4, 4>6, 6>9, 9>5

программа обнаруживает цикл вида: **2,5,9,6,4,3,2**.

* для системы отношений частичного порядка [1, с.648]:
* 9>2, 3>7, 7>5, 5>8, 8>6, 4>6, 1>3, 7>4, 9>5, 2>8, 1>9, 10>1, 6>10.

программа обнаруживает цикл вида: **2,9,1,10,6,8,2**.

***Замечания.***

1. *Если Вы загляните в [1, с.647], то учтите, что необходимо исправить следующие* ***неточности*** *в записи алгоритма на псевдокоде:*
   * *шаг* ***T9*** *алгоритма следует читать так: "Для 1<=k<=n установить P <- TOP[k], TOP[k] <- U и выполнить шаг T10.";*
   * *второе предложение в шаге T10 должно быть таким: "P <- NEXT(P)";*
   * *шаг T12 алгоритма следует читать так: "Установить TOP[k] <- W, где W - указатель, отличный от Nil, и k <-QLINK[k]. Теперь, если TOP[k] = Nil, повторить этот шаг.";*
   * *шаг T13 алгоритма следует читать так: "(Мы нашли начало цикла.) Напечатать значение k, установить TOP[k] <-Nil, k <- QLINK[k], и если TOP[k] = W, повторить этот шаг."*
2. *Алгоритм обнаружения контура при топологической сортировке изложен также в [2, с.495-496].*

(1) Кнут Д. Искусство программирования для ЭВМ. Т.1: Основные алгоритмы. - M.: Мир, 1976. - 736 с.

(2) Трамбле Ж., Соренсон П. Введение в структуры данных. - М.: Машиностроение, 1982. - 784 с.

На следующем шаге мы продолжим рассматривать ***примеры использования топологической сортировки***.

## Четвертый пример использования топологической сортировки

На этом шаге мы рассмотрим ***последний пример использования топологической сортировки***.

***Лемма [1, с.127]***.

В произвольном ориентированном бесконтурном графе вершины можно перенумеровать так, что каждая дуга будет иметь вид **(vi,vj)**, где **i<j**.

Для ***доказательства леммы*** предложим алгоритм, конструирующий такую нумерацию (в виде программы на языке **С++**).

Алгоритм основывается на следующем простом факте: в произвольном бесконтурном графе существует вершина, в которую не заходит ни одна дуга.

Чтобы убедиться в этом, выберем произвольную вершину **w1** графа, затем некоторую вершину **w2**, такую что **w1<-w2**, и т.д. Через конечное число шагов мы должны дойти до некоторой вершины **wi**, в которую не заходит ни одна дуга, ибо в силу бесконтурности ни одна вершина не может повторяться в последовательности **w1,w2,w3,...**

**#include** <iostream.h>

**#define** TRUE 1

**#define** FALSE 0

**typedef** **int** Boolean;

**// Нумерация вершин в бесконтурном графе.**

**typedef** **struct** Leader \*Lref; **//Тип: указатель на заголовочный узел.**

**typedef** **struct** Trailer \*Tref; **//Тип: указатель на дуговой узел.**

**//Описание типа заголовочного узла.**

**struct** Leader

{

**int** Key; **//Имя заголовочного узла.**

**int** Number; **//Новый номер вершины.**

**int** Count; **//Количество предшественников.**

**int** Count1; **//Количество последующих вершин.**

Tref Pred; **//Указатель на список предшественников.**

Tref Trail; **//Указатель на список последователей.**

Lref Next; **//Указатель на следующий узел в**

**//списке заголовочных узлов.**

};

**//Описание типа дугового узла.**

**struct** Trailer

{

Lref Id;

Tref Next;

};

**//Описание типа узла стека.**

**struct** Zveno

{

Lref Element; **//Указатель на вершину.**

Zveno\* Sled;

};

**class** Spisok {

**private:**

Lref Head; **//Указатель на список заголовочных узлов.**

Lref Tail; **//Указатель на фиктивный элемент**

**//в конце списка заголовочных узлов.**

Zveno\* Stack; **//Указатель на рабочий стек.**

Lref SearchGraph (**int**);

**void** Udalenie (Lref\*);

**void** V\_Stack (Lref);

**public:**

Spisok () {**//Инициализация списка заголовочных узлов.**

Head = Tail = **new** (Leader); };

**void** MakeGraph();

**void** PrintGraph();

**void** Renum();

**void** Vyvod();

};

**void** Spisok::MakeGraph ()

**//Функция возвращает указатель Head на структуру**

**//Вирта, соответствующую ориентированному графу.**

{

**int** x,y;

Lref p,q; **//Рабочие указатели.**

Tref t,r; **//Рабочие указатели.**

Boolean Res; **//Флаг наличия дуги.**

cout<<"Вводите начальную вершину дуги: ";

cin>>x;

**while** (x!=0)

{

cout<<"Вводите конечную вершину дуги: "; cin>>y;

**//Определим, существует ли в графе дуга (x,y)?**

p=SearchGraph (x); q=SearchGraph (y);

r = (\*p).Trail; Res = FALSE;

**while** ((r!=NULL)&&(!Res))

**if** ((\*r).Id==q) Res = TRUE;

**else** r = (\*r).Next;

**if** (!Res) **//Если дуга отсутствует, то поместим её в граф.**

{ t = **new** (Trailer); (\*t).Id = q;

(\*t).Next = (\*p).Trail; (\*p).Trail = t; (\*q).Count++;

t = **new** (Trailer); (\*t).Id = p;

(\*t).Next = (\*q).Pred; (\*q).Pred = t; (\*p).Count++; }

cout<<"Вводите начальную вершину дуги: "; cin>>x;

}

}

Lref Spisok::SearchGraph (**int** w)

**//Функция возвращает указатель на заголовочный узел**

**//с ключом w. Если заголовочный узел отсутствует, то он**

**//добавляется в список. Head - указатель на структуру Вирта.**

{

Lref h;

h = Head; (\*Tail).Key = w;

**while** ((\*h).Key!=w) h = (\*h).Next;

**if** (h==Tail)

**//В списке заголовочных узлов нет узла с ключом w.**

**//Поместим его в конец списка Head.**

{ Tail = **new** (Leader); (\*h).Count = (\*h).Count1 = 0;

(\*h).Trail = (\*h).Pred = **NULL**; (\*h).Next = Tail; }

**return** h;

}

**void** Spisok::PrintGraph ()

**//Вывод структуры Вирта, заданной указателем**

**//Head и соответствующей ориентированному графу.**

{

Lref p; **//Рабочий указатель.**

Tref q; **//Рабочий указатель.**

p = Head;

**while** (p!=Tail)

{

cout<<(\*p).Key<<"(("; q = (\*p).Trail;

**while** (q!=NULL)

{ cout<<(\*(\*q).Id).Key<<" "; q = (\*q).Next; }

cout<<")("; q = p->Pred;

**while** (q!=NULL)

{ cout << (\*(\*q).Id).Key<<" "; q = q->Next; }

cout << "))";

p = (\*p).Next; cout<<" ";

}

}

**void** Spisok::Vyvod()

{

Lref p = Head;

**while** ( p!=Tail )

{

cout << p->Key << '-'<< p->Number;

p = p->Next; cout << ' ';

}

}

**void** Spisok::Renum()

**//Перенумерация вершин графа, заданного структурой Вирта**

**//с указателем Head.**

{

**int** num;

Lref p,u;

Tref t;

Stack = **NULL**; p = Head;

**while** ( p!=Tail )

{

**if** ( p->Count!=0 ) V\_Stack(p);

p = p->Next;

}

num = 0;

**while** ( Stack!=NULL )

{

Udalenie (&u);

num++; u->Number = num;

t = u->Trail;

**while** ( t!=NULL )

{

t->Id->Count--;

**if** ( t->Id->Count==0 ) V\_Stack (t->Id);

t = t->Next;

}

}

}

**void** Spisok::Udalenie (Lref \*Klad)

**// Удаление элемента из стека Stack и сохранение его в**

**// переменной Klad.**

{

Zveno \*q;

**if** ( Stack==NULL )

cout << "Ошибка! Попытка выбоpа из пустого стека!\n";

**else**

{

(\*Klad) = Stack->Element; q = Stack;

Stack = Stack->Sled; **delete** q;

}

}

**void** Spisok::V\_Stack (Lref Elem)

**// Помещение Elem в стек Stack.**

{

Zveno \*q;

q = **new** (Zveno);

q->Element = Elem; q->Sled = Stack; Stack = q;

}

**void** main()

{

Spisok A;

**// Построение графа и вывод его структуры смежности.**

A.MakeGraph();

A.PrintGraph(); cout << endl;

A.Renum();

**// Вывод новых номеров вершин графа.**

A.Vyvod();

}

Текст этой программы можно взять [здесь](http://khpi-iip.mipk.kharkiv.edu/library/datastr/book_sod/kgsu/din92_1.zip).

Результат работы программы изображен на рисунке 1:

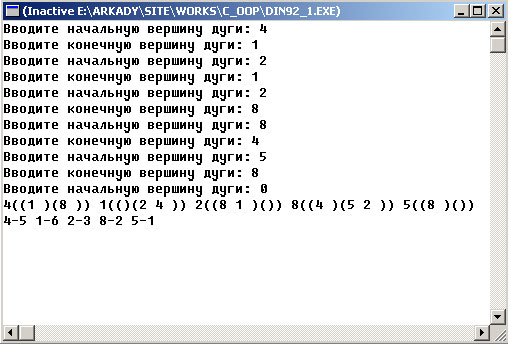


Рис.1. Результат работы приложения

(1) Липский В. Комбинаторика для программистов: Пер. с польск. - М.:Мир, 1988. - 213 с.

На следующем шаге мы дадим ***понятие о методе* PERT**.

## Понятие о методе PERT

На этом шаге мы приведем ***последний пример использования топологической сортировки***.

[***Сетевое планирование***](http://khpi-iip.mipk.kharkiv.edu/library/datastr/book_sod/kgsu/din_0131.html) - совокупность методов, использующих сетевую модель как основную форму представления информации об управляемом комплексе работ. Использование сетевого планирования позволяет повысить качетсво планирования и управления при реализации комплекса работ, например, дает возможность четко координировать деятельность всех сторон (организаций), участвующих в реализации, выделять наиболее важные задачи, определять сроки реализации, а также координировать план его реализации [1, с.540].

***Сетевая модель*** - информационная модель реализации некоторого комплекса взаимосвязанных работ, рассматриваемая как ориентированный граф без контуров, отображающий естественный порядок выполнения этих работ во времени; может содержать некоторые дополнительные характеристики (например, время, стоимость, ресурсы), относящиеся к отдельным работам и (или) к комплексу в целом.

Наибольшее распространение получило графическое представление сетевой модели на плоскости, называемое ***сетевым графиком*** [1, с.540].

Для быстрого "погружения" в постановку и решение ***задач сетевого планирования*** рассмотрим простейший пример, заимствованный нами из монографий [2; 3, с.409].

***Предположим, что шеф-повар получил заказ приготовить яичницу из одного яйца***. Вся процедура ее приготовления может быть разбита на ряд отдельных подзадач:

Взять яйцо ----> Разбить яйцо ----> Взять жир ---->

----> Положить жир на сковороду ----> Растопить жир ---->

----> Вылить яйцо на сковороду ---->

----> Ждать, пока яичница не изжарится ----> Снять яичницу

Некоторые из этих подзадач должны предшествовать другим (например, задача "взять яйцо" должна предшествовать задаче "разбить яйцо"). Ряд подзадач может выполняться параллельно (например, задачи "взять яйцо" и "растопить жир"). Шеф-повар хотел бы выполнить заказ как можно быстрее, при этом предполагается, что число его помощников не ограничено.

***Необходимо распределить работу среди помощников так, чтобы заказ был выполнен за минимально возможное время.***

Хотя этот пример может показаться легкомысленным, подобная задача возникает во многих ситуациях, связанных с планированием реальных действий. В больших вычислительных системах осуществляется планирование заданий с целью обеспечения минимального времени нахождения задания в системе, технолог на заводе может планировать организацию работы конвейера, минимизирующую время производства продукции, и т.п. Все проблемы подобного рода тесно связаны между собой и могут быть решены с использованием графов.

Давайте представим исходную задачу в виде ***графа***. Каждая вершина графа представляет собой подзадачу, а каждая дуга **(x,y)** представляет требование, что задача y не может выполняться до тех пор, пока не завершено выполнение задачи **x**. Граф задачи показан на рисунке:

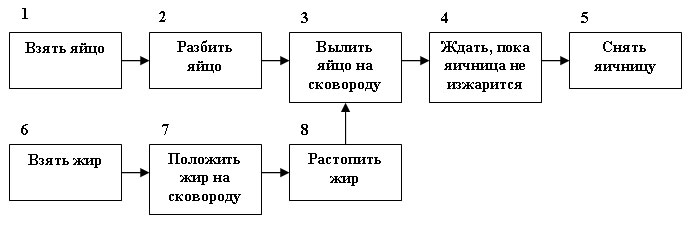


Рис.1. Граф задачи

Заметим, что в изображенном графе узлы 1 и 6 не имеют предшественников, и, следовательно, подзадачи, которые они представляют, могут выполняться сразу же и параллельно без ожидания завершения других подзадач. Все остальные подзадачи должны ждать завершения по крайней мере одной из этих подзадач. Как только эти первые две подзадачи завершены, соответствующие вершины и инцидентные дуги могут быть удалены из графа. Отметим, что получающийся в результате граф ***не содержит контуров***, поскольку вершины и дуги удалялись из графа без циклов. Стало быть, новый граф также должен содержать по крайней мере один узел, не имеющий предшественников. В нашем примере существуют два таких узла - 2 и 7. Значит, подзадачи 2 и 7 могут выполняться параллельно во второй период времени.

Продолжив построение далее, мы найдем, что минимальное время, за которое может быть поджарена яичница, -***шесть*** временных периодов (предполагая, что каждая подзадача требует ровно один период времени), а максимальное требующееся число помощников - два:

Период времени Помощник 1 Помощник 2

1 Взять яйцо Взять жир

2 Разбить яйцо Положить жир на сковороду

3 Растопить жир

4 Вылить яйцо на сковороду

5 Ждать, пока яичница не изжарится

6 Снять яичницу

Автоматизируем процесс построения решения задачи, модифицируя алгоритм топологической сортировки, проиллюстрированный программой из [89 шага](http://khpi-iip.mipk.kharkiv.edu/library/datastr/book_sod/kgsu/din_0089.html), следующим образом:

**while** (Граф не пуст)

{

Определить вершины, не имеющие предшественников.

Распечатать эту группу узлов с указанием, что эти подзадачи

могут быть выполнены параллельно в следующий момент времени.

Удалить из графа данные вершины и инцидентные дуги

}

Программа. ***Применение топологической сортировки для решения простейших задач сетевого планирования***.

**#include** <iostream.h>

**typedef** **struct** Leader \*Lref; **//Тип: указатель на заголовочный узел.**

**typedef** **struct** Trailer \*Tref; **//Тип: указатель на дуговой узел.**

**//Описание типа заголовочного узла.**

**typedef** **struct** Leader

{

**int** Key; **//Информационное поле.**

**int** Count; **//Количество предшественников.**

Tref Trail;

Lref Next;

};

**//Описание типа дугового узла.**

**typedef** **struct** Trailer

{

Lref Id;

Tref Next;

};

**class** Spisok {

**private**:

Lref Head; **//Указатель на список заголовочных узлов.**

Lref Tail; **//Указатель на фиктивный элемент**

**//в конце списка заголовочных узлов.**

**int** z; **//Количество узлов, не имеющих предшественников.**

**public**:

Spisok () {**//Инициализация списка заголовочных узлов.**

Head = Tail = **new** (Leader); z = 0;};

Lref L (**int**);

**void** Poisk();

**void** Vyvod();

};

**void** Spisok::Poisk()

{

Lref p,q; **// Рабочие указатели.**

p = Head; Head = **NULL**;

**while** (p!=Tail)

{

q = p; p = p->Next;

**if** (q->Count==0)

{ q->Next = Head; Head = q; }

}

}

Lref Spisok::L (**int** w)

**//Функция возвращает указатель на ведущего с ключом w.**

{

Lref h = Head;

Tail->Key = w;

**while** (h->Key!=w) h = h->Next;

**if** (h==Tail)

**// В списке нет элемента с ключом w.**

{

Tail = **new** (Leader); z++;

h->Count = 0; h->Trail = **NULL**; h->Next = Tail;

}

**return** h;

}

**void** Spisok::Vyvod()

{

Lref p,q; **// Рабочие указатели.**

Lref S,U; **// Рабочие указатели.**

Tref t;

cout << endl;

cout << "Результат...\n";

q = Head;

**while** ( q!=NULL)

**// Вывод всех элементов с нулевым количеством предшественников.**

{

S = q; cout << "( ";

**while** ( S!=NULL )

{ cout << S->Key << " "; z--; S = S->Next; }

cout << ") ";

**// ---------------------------------------------------**

U = **NULL**; **// Указатель на очередной список элементов**

**// с нулевым количеством предшественников.**

**while** ( q!=NULL )

{

t = q->Trail;

**while** ( t!=NULL )

{

p = t->Id; p->Count--;

**if** ( p->Count==0 ) **// Включение (\*p) в список ведущих.**

{ p->Next = U; U = p; }

t = t->Next;

}

q = q->Next;

}

q = U;

}

**if** (z!=0)

cout << "\nМножество не является частично упорядоченным!\n";

}

**void** main()

{

Spisok A;

Lref p,q; **// Рабочие указатели.**

Tref t;

**int** x,y; **// Рабочие переменные.**

**// Фаза ввода.**

cout << "Задайте отношение частичного порядка...\n";

cout << "Элемент ";

cin >> x;

cout << " предшествует элементу ";

**while** (x!=0)

{

cin >> y;

p = A.L(x); q = A.L(y);

t = **new** (Trailer); t->Id = q; t->Next = p->Trail;

p->Trail = t; q->Count += 1;

cout << "Элемент ";

cin >> x;

cout << " предшествует элементу ";

}

**// Поиск ведущих с нулевым количеством предшественников.**

A.Poisk();

**// Фаза вывода.**

A.Vyvod();

}

Текст этой программы можно взять [здесь](http://khpi-iip.mipk.kharkiv.edu/library/datastr/book_sod/kgsu/din93_1.zip).

Тестовые примеры:

* Решением задачи планирования приготовления яичницы является:
* ( 6 1 ) ( 2 7 ) ( 8 ) ( 3 ) ( 4 ) ( 5 )
* Решение задачи о топологическом упорядочении учебных курсов по ***математике, дискретной математике и информатике для студентов педагогических вузов*** ([тестовый пример 3 из 89 шага](http://khpi-iip.mipk.kharkiv.edu/library/datastr/book_sod/kgsu/din_0089.html#1)) имеет следующий вид:
* ( 26 22 21 19 1 ) ( 3 4 6 24 17 20 ) ( 2 7 5 )
* ( 8 18 10 ) ( 9 11 25 ) ( 27 12 14 16 ) ( 13 15 23 )

Интерпретируя результаты, получим ***перечень блоков учебных курсов***, которые можно преподавать ***параллельно***:

(Алгебра. Математический анализ. Геометрия. Математическая логика.

Дискретная математика);

(Теория автоматов. Машинная арифметика. Теория графов. Реляционная

алгебра и реляционное исчисление. Теория взаимодействующих последова-

тельных процессов. Теория алгоритмов);

(Теория кодирования. Формальные языки и грамматики. Введение в

программирование);

(Структуры данных. Устройство ЭВМ. Доказательство правильности

программ);

(Анализ алгоритмов. Компиляторы. Машинная графика);

(Операционные системы. Парадигмы программирования. Информационный

поиск. Вычислительная геометрия);

(Базы данных. Искусственный интеллект. Вычислительная математика).

***Замечание.******Дискретная математика*** *- область математики, занимающаяся изучением свойств* ***дискретных структур****, которые возникают как внутри математики, так и в ее приложениях. К числу таких структур могут быть отнесены, например, конечные группы, конечные графы, а также некоторые математические модели преобразователей информации, конечные автоматы, машины Тьюринга и т.д. Это примеры структур финитного (конечного) характера. Часть дискретной математики, изучающая их, называется* ***конечной математикой****. Иногда это понятие расширяют до дискретной математики. Помимо указанных финитных структур, дискретная математика изучает некоторые алгебраические системы, бесконечные графы, вычислительные схемы определенного вида, клеточные автоматы и т.д. [1, с.184].*

*Наряду с выделением дискретной математики путем указания ее предмета можно также определить дискретную математику посредством перечисления подразделов, составляющих дискретную математику. К ним в первую очередь должны быть отнесены*

* ***комбинаторный анализ****,*
* ***теория графов****,*
* ***теория кодирования****,*
* ***теория функциональных систем*** *и некоторые другие (см.[4, 5, 6, 7, 8, 9]).*

*Часто под термином* ***"дискретная математика"*** *(предполагая, что ее предмет исчерпывается конечными структурами) понимается именно совокупность перечисленных дисциплин.*

*Мы будем понимать термин "дискретная математика" именно так и отнесем к ней следующие* ***математические дисциплины****:*

* ***машинную арифметику****,*
* ***комбинаторный анализ****,*
* ***теорию кодирования****,*
* ***теорию графов****,*
* ***теорию автоматов****,*
* ***формальные языки и грамматики****,*
* ***теорию взаимодействующих последовательных процессов****.*

*Как отмечалось, возможно и более широкое толкование дискретной математики за счет расширения понимания ее предмета. С этой точки зрения к дискретной математике могут быть также отнесены как целые разделы математики, например,* ***математическая логика****, так и части таких разделов, как* ***теория чисел****,* ***алгебра****,* ***вычислительная математика****,* ***теория вероятностей*** *и другие, в которых изучаемый объект носит дискретный характер [1, с.184].*

(1) Математический энциклопедический словарь. - М.: Сов.энциклопедия, 1988. - 847 с.

(2) Tenenbaum A., Augenstein M. Data Structures Using Pascal. Englewood Cliffs. - N.Y.: Prentice-Hall, Inc. 1981.

(3) Лэнгсам Й., Огенстейн М., Тененбаум А. Структуры данных для персональных ЭВМ: Пер. с англ. - М.: Мир, 1989. - 568 с.

(4) Яблонский С.В. Введение в дискретную математику. - М.: Наука, 1986. - 384 с.

(5) Гаврилов Г.П., Сапоженко А.А. Сборник задач по дискретной математике. - М.: Наука, 1977. - 368 с.

(6) Горбатов В.А. Основы дискретной математики. - М.: Высш.шк., 1986. - 311 с.

(7) Горбатов В.А. Дискретная математика в задачах и упражнениях. - М.: МГИ, 1988 (1989). - 100 с.

(8) Триханов А.В. Основы дискретной математики. - Томск: ТПИ, 1987. - 96 с.

(9) Задачи по дискретной математике. - Алма-Ата: КазГУ, 1986. - 30 с.

На следующем шаге мы рассмотрим ***способы представления грамматики***.

## Представление грамматики

На этом шаге мы рассмотрим ***представление грамматики в памяти компьютера***.

Рассмотрим представление в памяти ***контекстно-свободной грамматики*** [1, с.455-457] с помощью ***плексов***.

***Плексы (сплетения)*** - структуры данных, служащие для представления графов, имеющих ребра, принадлежащих нескольким различным семействам [2, с.88]. ***И.Флорес*** [3, с.155-174] называет плексы ***ветвящимися списками***, а ***Д.Кнут*** [4, с.390-391,500-505] - ***Списками*** (с большой буквы!).

Отметим, что плексы широко применяются в ***трансляторах*** для отображения деревьев и графов в памяти ЭВМ, так как плексы лучше других типов структур отображают ***многоуровневые структуры данных*** в памяти ЭВМ.

Кстати, как заметил ***П.Холл*** [5, с.66], "... метод представления с помощью матрицы инцидентности - это разновидность метода плексов!"

Представление грамматики с помощью плексов становится удобным в тех случаях, когда грамматику необходимо рассматривать и обрабатывать путем ***нисходящего синтаксического анализа***. Эффективное представление грамматик в памяти необходимо в связи с тем, что различные варианты правил должны проверяться на их применимость на каждом шаге анализа конструкции фразы.

Представление синтаксиса с помощью ***графа*** имеет один существенный недостаток: компьютеры "***не умеют читать графы***", т.о. перед началом грамматического разбора нужно ***каким-то образом строить структуры данных***, представляющие грамматику.

Мы предполагаем, что грамматика представлена в виде детерминированного множества ***синтаксических графов***(***плексов***).

Естественный способ представить граф - это ввести узел для каждого символа и связать эти узлы при помощи ссылок. Узлы этой структуры представляют собой ***объединения***, один для ***терминального***, а другой для***нетерминального символа*** грамматики. Первый идентифицируется терминальным символом, который он обозначает, второй - ссылкой на структуру данных, представляющую соответствующий нетерминальный символ. Оба варианта содержат две ссылки: одна указывает на следующий символ, ***последователь*** (**Suc**), а другая связана со списком возможных ***альтернатив*** (**Alt**).

Теперь мы в состоянии привести описание соответствующего типа данных:

**typedef** **int** Boolean;

**typedef** **struct** Header \*Hpointer; **//Тип: указатель на заглавный узел.**

**typedef** **struct** Node

{

Node \*Suc;

Node \*Alt;

Boolean Terminal;

union

{

**char** Tsym [33];

Hpointer Nsym;

} TR;

};

Графически узел можно изобразить так:

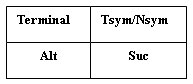


Рис.1. Графическое представление узла

Выясняется, что еще нужен элемент, представляющий пустую последовательность, символ "пусто". Мы обозначим его с помощью терминального элемента **Empty**.

Структура данных идентифицируется ***узлом-заголовком***, который содержит имя нетерминального символа (цели), к которому относится структура. Заголовок можно использовать для хранения выводимого на печать имени структуры:

**//Описание типа заглавного звена.**

**typedef** **struct** Header

{

Node\* Entry;

**char** Sym[33];

};

Пример [6,с.330,с.338]. Изобразим структуру для следующей грамматики:

A ::= x (B),

B ::= AC,

C ::= {+A}.

Здесь **+, x, (, )** - терминальные символы, а **{** и **}** - принадлежат расширенной БНФ (***форма Бэкуса-Наура***) и, следовательно, являются ***метасимволами***.

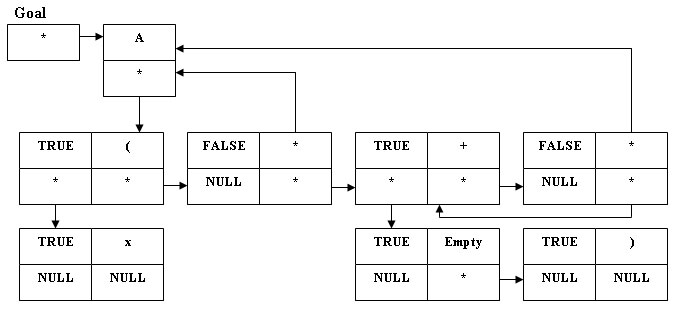


Рис.2. Грамматика

Ясно, что язык, порождаемый из приведенной грамматики, состоит из выражений с операндами **x**, знаком операции "+" и скобками.

Приведем примеры правильных предложений языка:

x

(x)

(x+x)

((x))

Программа, производящая ***грамматический разбор*** предложения, представленного в виде последовательности символов входного файла, состоит из повторяющегося оператора, описывающего переход от одного узла к следующему узлу [6,с.339]. Она оформлена как процедура, задающая интерпретацию графа; если встречается узел, представляющий нетерминальный символ, то интерпретация графа, на который он ссылается, предшествует завершению интерпретации текущего графа. Следовательно, процедура интерпретации вызывается ***рекурсивно***.

Если текущий символ (**Sym**) входного файла совпадает с символом в текущем узле структуры данных, то процедура переходит к узлу, на который указывает поле **Suc**, иначе - к узлу, на который указывает поле **Alt**:

**void** Spisok::Parse(Hpointer Goal, Boolean \*Match)

{

Node \*s;

s = Goal->Entry;

**do** {

**if** ( s->Terminal )

{

**if** (!strcmp(s->TR.Tsym,Sym))

{ (\*Match) = TRUE; Get\_Sym(); }

**else** (\*Match) = !(strcmp(s->TR.Tsym,"Empty"));

}

**else** Parse (s->TR.Nsym,&(\*Match));

**if** (\*Match) s = s->Suc;

**else** s = s->Alt;

}

**while** (s!=NULL);

}

**void** Spisok::Get\_Sym()

{

**char** Q;

cin >> Q;

Sym[0]=Q; Sym[1]='\0';

}

На основе функции **Parse** можно построить более сложные программы грамматического разбора, которые могут работать с более широкими классами грамматик.

Пример [6, с.339]. ***Программа грамматического разбора***. Программа написана ***И.А.Лебедевой*** (февраль 1993 г.).

**#include** <iostream.h>

**#include** <string.h>

**#define** TRUE 1

**#define** FALSE 0

**typedef** **int** Boolean;

**typedef** **struct** Header \*Hpointer; **//Тип: указатель на заглавный узел.**

**typedef** **struct** Node

{

Node \*Suc;

Node \*Alt;

Boolean Terminal;

union

{

**char** Tsym [33];

Hpointer Nsym;

} TR;

};

**//Описание типа заглавного звена.**

**typedef** **struct** Header

{

Node\* Entry;

**char** Sym[33];

};

**class** Spisok {

**private**:

Hpointer Goal; **//Указатель на заглавное звено.**

**char** Sym[33];

**void** Parse(Hpointer, Boolean \*);

**void** Get\_Sym();

**public**:

Spisok () { Goal = **NULL**; };

**void** Postr();

**void** Prov();

**int** Razbor();

};

**void** Spisok::Postr()

**// Построение синтаксического графа.**

{

Node \*UkZv; **//Текущий указатель.**

Node \*UkZv1; **//Текущий указатель.**

Goal = **new** (Header);

strcpy(Goal->Sym,"A");

**// Goal +-----+**

**// \* ---> | A |**

**// +-----+**

**// | |**

**// +-----+**

**// ------------------------------**

Goal->Entry = **new** (Node);

UkZv = Goal->Entry;

UkZv->Terminal = TRUE;

strcpy(UkZv->TR.Tsym,"(");

**// Goal +-----+**

**// \* ---> | A |**

**// +-----+**

**// | \* |**

**// +--|--+**

**// |**

**// V**

**// +----+----+**

**// UkZv |TRUE| ( |**

**// \* ---> +----+----+**

**// | | |**

**// +----+----+**

**// ------------------------------**

UkZv->Alt = **new** (Node);

UkZv1 = UkZv->Alt; UkZv1->Terminal = TRUE;

strcpy(UkZv1->TR.Tsym,"x"); UkZv1->Alt = UkZv1->Suc = **NULL**;

**// Goal +-----+**

**// \* ---> | A |**

**// +-----+**

**// | \* |**

**// +--|--+**

**// |**

**// V**

**// +----+----+**

**// UkZv |TRUE| ( |**

**// \* ---> +----+----+**

**// | \* | |**

**// +-|--+----+**

**// |**

**// V**

**// +----+----+**

**// UkZv1 |TRUE| x |**

**// \* ---> +----+----+**

**// |NULL|NULL|**

**// +----+----+**

**// ------------------------------**

UkZv->Suc = **new** (Node);

UkZv = UkZv->Suc; UkZv->Terminal = FALSE;

UkZv->TR.Nsym = Goal; UkZv->Alt = **NULL**;

**// Goal +-----+**

**// \* ---> | A | <---------------+**

**// +-----+ |**

**// | \* | UkZv \* |**

**// +--|--+ | |**

**// | | |**

**// V V |**

**// +-----+-----+ +-----+--|--+**

**// |TRUE | ( | |FALSE| \* |**

**// +-----+-----+ +-----+-----+**

**// | \* | \* ---->|NULL | |**

**// +--|--+-----+ +-----+-----+**

**// |**

**// V**

**// +-----+-----+**

**// UkZv1 |TRUE | x |**

**// \* ---> +-----+-----+**

**// |NULL |NULL |**

**// +-----+-----+**

**// -----------------------------------------**

UkZv->Suc = **new** (Node);

UkZv1 = UkZv->Suc; UkZv1->Terminal = TRUE;

strcpy(UkZv1->TR.Tsym,"+");

**// Goal +-----+**

**// \* ---> | A | <---------------+**

**// +-----+ |**

**// | \* | UkZv \* | UkZv1 \***

**// +--|--+ | | |**

**// | | | |**

**// V V | V**

**// +-----+-----+ +-----+--|--+ +-----+-----+**

**// |TRUE | ( | |FALSE| \* | |TRUE | + |**

**// +-----+-----+ +-----+-----+ +-----+-----+**

**// | \* | \* ---->|NULL | \* ---->| | |**

**// +--|--+-----+ +-----+-----+ +-----+-----+**

**// |**

**// V**

**// +-----+-----+**

**// |TRUE | x |**

**// +-----+-----+**

**// |NULL |NULL |**

**// +-----+-----+**

**// ------------------------------------------------------**

UkZv1->Suc = **new** (Node); UkZv1 = UkZv1->Suc;

UkZv = UkZv->Suc; UkZv1->Terminal = FALSE;

UkZv1->TR.Nsym = Goal; UkZv1->Alt = **NULL**; UkZv1->Suc = UkZv;

**//Goal +-----+**

**// \* ->| A | <---------------------------------------------+**

**// +-----+ <---------------+ |**

**// | \* | | UkZv \* UkZv1 \* |**

**// +--|--+ | | | |**

**// | | | | |**

**// V | V V |**

**// +-----+-----+ +-----+--|--+ +-----+-----+ +-----+--|--+**

**// |TRUE | ( | |FALSE| \* | |TRUE | + | |FALSE| \* |**

**// +-----+-----+ +-----+-----+ +-----+-----+ +-----+-----+**

**// | \* | \* --->|NULL | \* --->| | \* --->|NULL | \* |**

**// +--|--+-----+ +-----+-----+ +-----+-----+ +-----+--|--+**

**// | ^ |**

**// V +-----------------+**

**// +-----+-----+**

**// |TRUE | x |**

**// +-----+-----+**

**// |NULL |NULL |**

**// +-----+-----+**

**// --------------------------------------------------------------**

UkZv->Alt = **new** (Node);

UkZv = UkZv->Alt; UkZv->Terminal = TRUE;

strcpy(UkZv->TR.Tsym,"Empty"); UkZv->Alt = **NULL**;

**//Goal +-----+**

**// \* ->| A | <---------------------------------------------+**

**// +-----+ <---------------+ |**

**// | \* | | UkZv1 \* |**

**// +--|--+ | | |**

**// | | | |**

**// V | V |**

**// +-----+-----+ +-----+--|--+ +-----+-----+ +-----+--|--+**

**// |TRUE | ( | |FALSE| \* | |TRUE | + | |FALSE| \* |**

**// +-----+-----+ +-----+-----+ +-----+-----+ +-----+-----+**

**// | \* | \* --->|NULL | \* --->| \* | \* --->|NULL | \* |**

**// +--|--+-----+ +-----+-----+ +--|--+-----+ +-----+--|--+**

**// | | ^ |**

**// | | +-----------------+**

**// V V**

**// +-----+-----+ +-----+-----+**

**// |TRUE | x | UkZv |TRUE |Empty|**

**// +-----+-----+ \* --> +-----+-----+**

**// |NULL |NULL | |NULL | |**

**// +-----+-----+ +-----+-----+**

**// --------------------------------------------------------------**

UkZv->Suc = **new** (Node); UkZv = UkZv->Suc;

UkZv->Terminal = TRUE;

strcpy(UkZv->TR.Tsym,")"); UkZv->Alt = UkZv->Suc = **NULL**;

**//Goal +-----+**

**// \* ->| A | <---------------------------------------------+**

**// +-----+ <---------------+ |**

**// | \* | | UkZv1 \* |**

**// +--|--+ | | |**

**// V | V |**

**// +-----+-----+ +-----+--|--+ +-----+-----+ +-----+--|--+**

**// |TRUE | ( | |FALSE| \* | |TRUE | + | |FALSE| \* |**

**// +-----+-----+ +-----+-----+ +-----+-----+ +-----+-----+**

**// | \* | \* --->|NULL | \* --->| \* | \* --->|NULL | \* |**

**// +--|--+-----+ +-----+-----+ +--|--+-----+ +-----+--|--+**

**// | | ^ |**

**// | | +-----------------+**

**// V V**

**// +-----+-----+ +-----+-----+ +-----+-----+**

**// |TRUE | x | |TRUE |Empty| |TRUE | ) |**

**// +-----+-----+ +-----+-----+ +-----+-----+**

**// |NULL |NULL | |NULL | \* --->|NULL |NULL |**

**// +-----+-----+ +-----+-----+ +-----+-----+**

**// ^**

**// |**

**// UkZv \***

}

**void** Spisok::Prov()

**// Проверка графа.**

{

Node \*A, \*C; **//Рабочие переменные.**

cout << "Элементарная проверка построенного графа...\n";

A = Goal->Entry; cout << A->TR.Tsym << " ";

cout << A->Suc->Suc->TR.Tsym << " ";

cout << A->Suc->Suc->Alt->Suc->TR.Tsym << " ";

cout << A->Alt->TR.Tsym << " ";

C = A->Suc->TR.Nsym->Entry; cout << C->TR.Tsym << endl;

cout << "Проверка окончена!\n";

}

**int** Spisok::Razbor()

**// Синтаксический разбор.**

{

Boolean Rez = FALSE;

**// Приступим к грамматическому разбору.**

cout << "Вводите строку...\n";

cout << "Конец ввода - символ \"точка\".\n";

Get\_Sym();

Parse(Goal,&Rez); cout << endl;

**if** ((Rez) && !strcmp(Sym,".")) **return** 1;

**else** **return** 0;

}

**void** Spisok::Get\_Sym()

{

**char** Q;

cin >> Q;

Sym[0]=Q; Sym[1]='\0';

}

**void** Spisok::Parse(Hpointer Goal, Boolean \*Match)

{

Node \*s;

s = Goal->Entry;

**do** {

**if** ( s->Terminal )

{

**if** (!strcmp(s->TR.Tsym,Sym))

{ (\*Match) = TRUE; Get\_Sym(); }

**else** (\*Match) = !(strcmp(s->TR.Tsym,"Empty"));

}

**else** Parse (s->TR.Nsym,&(\*Match));

**if** (\*Match) s = s->Suc;

**else** s = s->Alt;

}

**while** (s!=NULL);

}

**void** main()

{

Spisok A;

**// Построение синтаксического графа.**

A.Postr();

A.Prov();

**if** (A.Razbor()) cout << "Все хорошо!";

**else** cout << "Ошибка!";

}

Текст этой программы можно взять [здесь](http://khpi-iip.mipk.kharkiv.edu/library/datastr/book_sod/kgsu/din94_1.zip).

(1) Трамбле Ж., Соренсон П. Введение в структуры данных. - М.: Машиностроение, 1982. - 784 с.

(2) Разумов О.С. Организация данных в вычислительных системах. - М.: Статистика, 1978. - 184 с.

(3) Флорес И. Структуры и управление данными. - М.: Финансы и статистика, 1982. - 319 с.

(4) Кнут Д. Искусство программирования для ЭВМ. Т.1: Основные алгоритмы. - M.: Мир, 1976. - 736 с.

(5) Холл П. Вычислительные структуры. Введение в нечисленное программирование. - М.: Мир, 1978. - 214 с.

(6) Вирт H. Алгоритмы + структуры данных = программы. - М.: Мир, 1985. - 406 с.

Со следующего шага мы начнем знакомиться со ***способами обхода графов***.

## Обход графов (общие сведения)

На этом шаге мы приведем ***общие сведения по обходу графов***.

Существует много алгоритмов на графах, в основе которых лежит систематический перебор вершин графа, такой что каждая вершина просматривается (***посещается***) в точности ***один раз***. Поэтому важной задачей является нахождение хороших методов поиска в графе.

Под ***обходом графов*** (***поиском на графах***) мы будем понимать процесс систематического просмотра всех вершин графа с целью отыскания вершин, удовлетворяющих некоторому условию.

***В.Липский*** [1, с.88] называет метод поиска "***хорошим***", если

* он позволяет алгоритму решения интересующей нас задачи легко "погрузиться" в этот метод и
* каждое ребро графа анализируется не более одного раза (или, что существенно не меняет ситуации, число раз, ограниченное константой).

Два наиболее распространенных алгоритма обхода графов называются:

* [***обход графа в глубину***](http://khpi-iip.mipk.kharkiv.edu/library/datastr/book_sod/kgsu/din_0096.html) (***поиск в глубину*** (англ. **Depth First Search**)) и
* [***обход графа в ширину***](http://khpi-iip.mipk.kharkiv.edu/library/datastr/book_sod/kgsu/din_0097.html) (***поиск в ширину*** (англ. **Breadth First Search**)).

(1) Липский В. Комбинаторика для программистов: Пер. с польск. - М.: Мир, 1988. - 213 с.

На следующем шаге мы рассмотрим ***обход графа в глубину***.

## Обход графов в глубину

На этом шаге мы рассмотрим ***алгоритм обхода графа в глубину***.

***Обход в глубину*** (называемый иногда ***стандартным обходом***), есть обход графа по следующим правилам:

* находясь в вершине **x**, нужно двигаться в любую другую, ранее не посещенную вершину (если таковая найдется), одновременно запоминая дугу, по которой мы впервые попали в данную вершину;
* если из вершины **x** мы не можем попасть в ранее не посещенную вершину или таковой вообще нет, то мы возвращаемся в вершину **z**, из которой впервые попали в **x**, и продолжаем обход в глубину из вершины **z**.

При выполнении обхода графа по этим правилам мы стремимся проникнуть "вглубь" графа так далеко, как это возможно, затем отступаем на шаг назад и снова стремимся пройти вперед и так далее.

На этом шаге мы приведем рекурсивную функцию обхода графа в глубину. Граф представлен в памяти структурой Вирта.

Заметим, что перед обращением к функции **Depth\_First\_Search ()** необходимо провести инициализацию:

t = Head;

**while** (t!=Tail)

{(\*t).Flag = TRUE; t = (\*t).Next;}

Текст функции:

**void** Depth\_First\_Search (Lref r)

{

Tref t;

t = (\*r).Trail; cout<<(\*r).Key;

(\*r).Flag = FALSE;

**while** (t!=**NULL**)

{ **if** ((\*(\*t).Id).Flag)

Depth\_First\_Search ((\*t).Id);

t = (\*t).Next; }

}

В связи с тем, что поиск в глубину играет важную роль в проектировании алгоритмов на графах, представим также нерекурсивную версию функции **Depth\_First\_Search()**, которую мы назовем **Depth\_First\_Search\_1()**.

В приведенной ниже формулировке нерекурсивного алгоритма поиска в глубину на графе [1,с.125-126] предполагается: во-первых, что зафиксирован некоторый линейный порядок на множестве всех вершин графа, и, во-вторых, что множество вершин, смежных со всякой вершиной графа, также линейно упорядочено.

**while** (Имеется хотя бы одна непосещенная вершина)

{

Пусть p - первая из непосещенных вершин.

Посетить вершину p и поместить ее в пустой стек S;

**while** ( Стек S непуст )

{

Пусть p - вершина, находящаяся на верхушке стека S;

**if** (У вершины p есть непосещенные смежные вершины)

{ Пусть q - первая непосещенная вершина из вершин, смежных

вершине p. Пройти по ребру (p,q), посетить вершину q и

поместить ее в стек S }

**else** Удалить вершину p из стека S

}

}

Реализуем ту часть приведенного алгоритма, которая ограничивается обходом только одной связной компоненты графа (смотри [шаг 78](http://khpi-iip.mipk.kharkiv.edu/library/datastr/book_sod/kgsu/din_0078.html#1)).

Не забудьте про предварительную инициализацию:

t = Head;

**while** (t!=Tail)

{ (\*t).Flag = TRUE; t = (\*t).Next; }

Текст функции:

**void** Depth\_First\_Search\_1 (Lref r)

{

Tref t;

svqz Stack;

Stack = **NULL**; **//Стек пуст.**

**//Посетим первую непосещенную вершину графа и**

**//поместим ее указатель на ее список смежности**

**//в первоначально пустой стек.**

cout<<(\*r).Key; (\*r).Flag = FALSE;

W\_S (&Stack,(\*r).Trail);

**while** (Stack!=**NULL**)

{ **//Рассмотрим "верхушку" стека.**

t = (\*Stack).Element;

**if** ((\*(\*t).Id).Trail!=**NULL**) {

**//У рассматриваемой вершины есть смежные.**

**if** ((\*(\*t).Id).Flag) **//У рассматриваемой вершины есть**

**// непосещенные смежные вершины.**

{ **//Посетим рассматриваемую вершину и поместим**

**//указатель на ее список смежности в стек.**

cout<<(\*(\*t).Id).Key);

(\*(\*t).Id).Flag = FALSE; W\_S (&Stack,(\*(\*t).Id).Trail); }

**//У рассматриваемой вершины нет**

**// непосещенных смежных вершин.**

**else** { t = (\*Stack).Element;

**if** ((\*t).Next!=**NULL**) {

**//Заменяем верхушку стека**

**// указателем на следующий элемент списка смежности.**

YDALENIE (&Stack,&t); W\_S (&Stack,(\*t).Next); }

**//Удаляем верхушку стека.**

**else** YDALENIE (&Stack,&t); }

}

**//У рассматриваемой вершины нет смежных вершин.**

**else** {

**if** ((\*(\*t).Id).Flag) {

**//Посетим рассматриваемую вершину.**

cout<<(\*(\*t).Id).Key;

(\*(\*t).Id).Flag = FALSE; }

t = (\*Stack).Element;

**if** ((\*t).Next!=**NULL**) { **//Заменяем верхушку стека указателем на**

**//следующий элемент списка смежности.**

YDALENIE (&Stack,&t); W\_S (&Stack,(\*t).Next); }

**//Удаляем верхушку стека.**

**else** YDALENIE (&Stack,&t); }

}

}

Пример. Демонстрация рекурсивного и нерекурсивного обходов графа в глубину. Граф представлен структурой Вирта.

**#include** <iostream.h>

**#define** TRUE 1

**#define** FALSE 0

**typedef** **int** Boolean;

**typedef** **struct** L \*Lref; **// Тип: указатель на заголовочный узел.**

**typedef** **struct** T \*Tref; **// Тип: указатель на дуговой узел.**

**//Описание типа заголовочного узла.**

**typedef** **struct** L

{

**int** Key; **//Имя заголовочного узла.**

**int** Count; **//Количество предшественников.**

Boolean Flag; **//Флаг посещения узла при обходе.**

Tref Trail; **//Указатель на список смежности.**

Lref Next; **//Указатель на следующий узел в списке заголовочных узлов.**

} Leader;

**//Описание типа дугового узла.**

**typedef** **struct** T

{

Lref Id;

Tref Next;

} Trailer;

**//Описание типа узла стека.**

**typedef** Tref TipElement;

**typedef** **struct** Zveno \*svqz;

**typedef** **struct** Zveno

{

TipElement Element; **//Указатель на список смежности.**

svqz Sled;

} St;

**class** Spisok

{

**private**:

Lref Head; **//Указатель на голову списка заголовочных узлов.**

Lref Tail; **//Указатель на фиктивный элемент**

**// в конце списка заголовочных узлов.**

**void** SearchGraph (int, Lref \*);

**void** W\_S (svqz \*, TipElement);

**void** YDALENIE (svqz \*, TipElement \*);

**public**:

Spisok() {**//Инициализация списка заголовочных узлов.**

Head = Tail = **new** (Leader); }

Lref GetHead() { **return** Head; }

Lref GetTail() { **return** Tail; }

**void** MakeGraph ();

**void** PrintGraph ();

**void** Depth\_First\_Search (Lref);

**void** Depth\_First\_Search\_1 (Lref);

};

**void** main ()

{

Spisok A;

Lref t; **//Рабочий указатель для перемещения**

**// по списку заголовочных звеньев.**

**//Построение графа и вывод его структуры Вирта.**

A.MakeGraph ();

A.PrintGraph (); cout<<endl;

**//Рекурсивный обход графа в глубину.**

cout<<"Результат рекурсивного обхода...\n";

t = A.GetHead();

**while** (t!=A.GetTail())

{ (\*t).Flag = TRUE; t = (\*t).Next; }

A.Depth\_First\_Search (A.GetHead()); cout<<endl;

**//Нерекурсивный обход графа в глубину.**

cout<<"Результат нерекурсивного обхода...\n";

t = A.GetHead();

**while** (t!=A.GetTail())

{ (\*t).Flag = TRUE; t = (\*t).Next;}

A.Depth\_First\_Search\_1 (A.GetHead()); cout<<endl;

}

**void** Spisok::SearchGraph (**int** w, Lref \*h)

**//Функция возвращает указатель на заголовочный узел**

**//с ключом w в графе, заданном структурой Вирта с указателем Head.**

{

\*h = Head; (\*Tail).Key = w;

**while** ((\*\*h).Key!=w) \*h = (\*\*h).Next;

**if** (\*h==Tail)

**//В списке заголовочных узлов нет узла с ключом w.**

**//Поместим его в конец списка Head.**

{ Tail = **new** (Leader); (\*\*h).Count = 0;

(\*\*h).Trail = **NULL**; (\*\*h).Next = Tail; }

}

**void** Spisok::MakeGraph ()

**//Функция возвращает указатель Head на структуру**

**//Вирта, соответствующую ориентированному графу.**

{

**int** x,y;

Lref p,q; **//Рабочие указатели.**

Tref t,r; **//Рабочие указатели.**

Boolean Res; **//Флаг наличия дуги.**

cout<<"Вводите начальную вершину дуги: ";

cin>>x;

**while** (x!=0)

{

cout<<"Вводите конечную вершину дуги: "; cin>>y;

**//Определим, существует ли в графе дуга (x,y)?**

SearchGraph (x, &p); SearchGraph (y,&q);

r = (\*p).Trail; Res = FALSE;

**while** ((r!=**NULL**)&&(!Res))

**if** ((\*r).Id==q) Res = TRUE;

**else** r = (\*r).Next;

**if** (!Res) **//Если дуга отсутствует, то поместим её в граф.**

{ t = **new** (Trailer); (\*t).Id = q;

(\*t).Next = (\*p).Trail; (\*p).Trail = t; (\*q).Count++; }

cout<<"Вводите начальную вершину дуги: "; cin>>x;

}

}

**void** Spisok::PrintGraph ()

**//Вывод структуры Вирта, заданной указателем**

**//Head и соответствующей ориентированному графу.**

{

Lref p; **//Рабочий указатель.**

Tref q; **//Рабочий указатель.**

p = Head;

**while** (p!=Tail)

{

cout<<"("<<(\*p).Key; q = (\*p).Trail;

**while** (q!=**NULL**)

{ cout<<(\*(\*q).Id).Key; q = (\*q).Next; }

cout<<")"; p = (\*p).Next; cout<<" ";

}

}

**void** Spisok::W\_S (svqz \*stk, TipElement el)

**//Помещение элемента el в стек stk.**

{

svqz q=new (St);

(\*q).Element = el; (\*q).Sled = \*stk; \*stk = q;

}

**void** Spisok::YDALENIE (svqz \*stk, TipElement \*klad)

**//Удаление звена из стека, заданного указателем \*stk.**

**//Значение информационного поля удаляемого звена сохраня-**

**//ется в параметре klad.**

{

svqz q;

**if** (\*stk==**NULL**) cout<<"Попытка выбора из пустого стека!\n";

**else**

{ \*klad = (\*\*stk).Element; q = \*stk; \*stk = (\*\*stk).Sled; **delete** q; }

}

**void** Spisok::Depth\_First\_Search (Lref r)

**//Рекурсивный обход графа в глубину. r - указатель**

**//на структуру Вирта.**

{

Tref t;

t = (\*r).Trail; cout<<(\*r).Key; (\*r).Flag = FALSE;

**while** (t!=**NULL**)

{ **if** ((\*(\*t).Id).Flag) Depth\_First\_Search ((\*t).Id); t = (\*t).Next; }

}

**void** Spisok::Depth\_First\_Search\_1 (Lref r)

**//Нерекурсивный обход графа в глубину.**

**//r - указатель на структуру Вирта.**

{

Tref t;

svqz Stack = **NULL**; **//Стек пуст.**

**//Посетим первую непосещенную вершину графа и**

**//поместим ее указатель на ее список смежности**

**//в первоначально пустой стек.**

cout<<(\*r).Key; (\*r).Flag = FALSE;

W\_S (&Stack,(\*r).Trail);

**while** (Stack!=**NULL**)

{ **//Рассмотрим "верхушку" стека.**

t = (\*Stack).Element;

**if** ((\*(\*t).Id).Trail!=**NULL**)

{ **//У рассматриваемой вершины есть смежные вершины.**

**if** ((\*(\*t).Id).Flag)

**//У рассматриваемой вершины есть**

**// непосещенные смежные вершины.**

{

**//Посетим рассматриваемую вершину**

**// и поместим указатель на ее список смежности в стек.**

cout<< (\*(\*t).Id).Key; (\*(\*t).Id).Flag = FALSE;

W\_S (&Stack,(\*(\*t).Id).Trail);

}

**//У рассматриваемой вершины нет**

**//непосещенных смежных вершин.**

**else** {

t = (\*Stack).Element;

**if** ((\*t).Next!=**NULL**)

**//Заменяем верхушку стека**

**//указателем на следующий элемент списка смежности.**

{ YDALENIE (&Stack,&t); W\_S (&Stack,(\*t).Next); }

**//Удаляем верхушку стека.**

**else** YDALENIE (&Stack,&t);

}

}

**//У рассматриваемой вершины нет смежных вершин.**

**else** {

**if** ((\*(\*t).Id).Flag) **//Посетим рассматриваемую вершину.**

{ cout<<(\*(\*t).Id).Key; (\*(\*t).Id).Flag = FALSE; }

t = (\*Stack).Element;

**if** ((\*t).Next!=**NULL**)

**//Заменяем верхушку стека указателем на следующий**

**// элемент списка смежности.**

{ YDALENIE (&Stack,&t); W\_S (&Stack,(\*t).Next); }

**//Удаляем верхушку стека.**

**else** YDALENIE (&Stack,&t);

}

}

}

Текст этой программы можно взять [здесь](http://khpi-iip.mipk.kharkiv.edu/library/datastr/book_sod/kgsu/din96_1.zip).

Результат работы приложения приведен на рисунке 1:

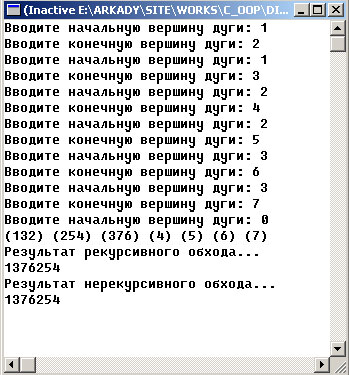


Рис.1. Результат работы приложения

***Замечание [2, с.91]****. Методика обхода в глубину очевидным образом переносится на ориентированные графы. Нетрудно проверить, что в случае ориентированного графа результатом вызова функций* ***Depth\_First\_Search()*** *и* ***Depth\_First\_Search\_1()*** *будет посещение всех вершин* ***u****, таких что существует путь из* ***v*** *в* ***u****. В самом деле, при обходе в глубину ориентированного графа мы можем попасть в вершину* ***y*** *из вершины* ***x*** *только в том случае, если в графе есть дуга* ***(x,y)****, т.е. мы должны двигаться вперед в направлении ориентации дуг, а возвращаться против ориентации (конечно же, в неориентированном графе таких ограничений нет)!*

Функции языка **LISP**, реализующие ***алгоритм обхода графа в глубину***, выглядят следующим образом [3, с.128].

Пример 2.

(DEFUN TESTDF (LAMBDA NIL

(PRINT "Построение графа.")

(SETQ GRAPH NIL) **;Инициализация**

(PRINT "Введите список вершин графа:") (SETQ NODE (READ))

(PRINT "Введите список списков смежных вершин:")

(SETQ NODELIST (READ))

(PRINT (SETQ GRAPH (PAIRLIS NODE NODELIST GRAPH)))

(PRINT "Приступим к обходу графа в глубину...")

(PRINT "Введите вершину графа, с которой начнется обход:")

(SETQ ROOT (READ))

(DEPTHFIRST GRAPH ROOT)

))

**; ------------------------------------**

(DEFUN PAIRLIS (LAMBDA (KEY ADJ GRAPH)

**; Построение графа из списка вершин KEY и списка списков**

**; смежных вершин ADJ путем добавления к существующему**

**; графу GRAPH**

(COND ( (**NULL** KEY) GRAPH )

( (**NULL** ADJ) GRAPH )

( T (CONS (CONS (CAR KEY) (CAR ADJ))

(PAIRLIS (CDR KEY) (CDR ADJ) GRAPH)) )

)

))

**; ------------------------------------**

(DEFUN DEPTHFIRST (LAMBDA (GRAPH ROOT)

**; Обход графа в глубину:**

**; GRAPH - граф, представленный структурой смежности в**

**; виде ассоциативного списка,**

**; ROOT - вершина, с которой начинается обход графа,**

**; Результат: список вершин графа в порядке посещения в глубину**

(COND ( (**NULL** GRAPH) NIL )

( T (DEFI GRAPH (LIST ROOT) (LIST ROOT)) )

)

))

**; --------------------------------------**

(DEFUN DEFI (LAMBDA (GRAPH VISITED PATH)

**; GRAPH - граф, представленный структурой смежности в**

**; виде ассоциативного списка,**

**; VISITED - список уже посещенных вершин,**

**; PATH - список вершин, определяющий путь посещения**

(COND

( (**NULL** PATH) (REVERSE VISITED) )

( T (COND ( (**NULL** (EXPND GRAPH VISITED (CAR PATH)))

(DEFI GRAPH VISITED (CDR PATH)) )

( T (DEFI GRAPH

(CONS (EXPND GRAPH VISITED

(CAR PATH))

VISITED)

(CONS (EXPND GRAPH VISITED

(CAR PATH))

PATH)) )) )

)

))

**; -----------------------------------------**

(DEFUN EXPND (LAMBDA (GRAPH VISITED VERTEX)

**; Выбор в графе GRAPH следующей еще не просмотренной**

**; вершины, смежной с вершиной VERTEX**

(COND ( (**NULL** (NEIGHBOUR3 VERTEX GRAPH)) NIL )

( T (FIRSTNOTVISITED

VISITED

(NEIGHBOUR3 VERTEX GRAPH))

)

)

))

**; --------------------------------------------**

(DEFUN FIRSTNOTVISITED (LAMBDA (VISITED VLIST)

**; Поиск первой непосещенной вершины в списке VLIST**

**; VISITED - список уже посещенных вершин**

(COND ( (**NULL** VLIST) NIL )

( T (COND ( (**NULL** (MEMBER (CAR VLIST) VISITED))

(CAR VLIST)

)

( T (FIRSTNOTVISITED

VISITED

(CDR VLIST)) )) )

)

))

**; ---------------------------------**

(DEFUN NEIGHBOUR3 (LAMBDA (X GRAPH)

**; Функция возвращает список вершин графа GRAPH, смежных с**

**; вершиной X**

(COND ( (**NULL** (ASSOC X GRAPH)) NIL )

( T (CDR (ASSOC X GRAPH)) )

)

))

Текст этой библиотеки можно взять [здесь](http://khpi-iip.mipk.kharkiv.edu/library/datastr/book_sod/kgsu/din96_2.zip).

Тестовые примеры:

1) $ (TESTDF)

Построение графа.

Введите список вершин графа:

(1 2 3 4)

Введите список списков смежных вершин:

((2 3 4) (3 1) (4) ())

Приступим к обходу графа в глубину...

Введите вершину графа, с которой начнется обход:

2

(2 3 4 1)

2) $ (TESTBF)

Построение графа.

Введите список вершин графа:

(1 2 3 4 5 6 7)

Введите список списков смежных вершин:

((2 3) (4 5) (6 7) () () () ())

Приступим к обходу графа в глубину...

Введите вершину графа, с которой начнется обход:

1

(1 2 3 4 5 6 7)

Последний тестовый пример иллюстрируется следующим ориентированным графом:

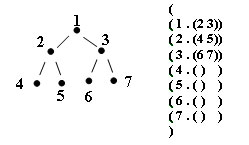


Рис.2. Пример графа

Программа работает следующим образом.

Если граф не пуст, то в два списка: **VISITED** - список уже посещенных вершин и **PATH** - список вершин, определяющих путь посещения, заносится первая вершина графа и после этого вызывается функция **DEFI**. Ее третий аргумент - список **PATH** - позволяет нам в любой момент вернуться к предыдущей вершине. Список **VISITED**используется для того, чтобы помнить о том, какие вершины уже посещались.

Выбор следующей вершины осуществляется с помощью функции **EXPND**. Именно работой этой функции определяется порядок обхода графа. В нашем случае, пока возможно, выбирается смежная вершина, т.е. на каждом шаге алгоритма делается попытка пройти ***"вглубь"*** графа. В противном случае из списка **PATH** удаляется первый элемент, и поиск возобновляется с предыдущей вершины.

Функция **DEPTHFIRST** возвращает список вершин графа в том порядке, в котором эти вершины посещались. Очевидно, что этот порядок зависит от вершины, с которой начинается просмотр.

***Замечания.***

1. *Алгоритмы поиска в глубину на графе изложены в монографиях [4, с.361-364], [2, с.88-91], [5, с.198-205], [6, с.323-327].*
2. *Техника обхода в глубину использовалась в алгоритмах на графах долгое время, однако степень, с которой эта процедура позволяет раскрыть свойства графа, была обнаружена недавно. Первой работой в этом направлении является статья Р.Е.Тарьяна, опубликованная в 1972 г. (ссылку см.в [4, с.426]). В этой работе представлена основная процедура поиска в глубину [4, с.364] и алгоритмы отыскания двусвязных [4, с.371] и сильно связных компонент графа [4,c.375].*
3. ***M-нумерацией*** *вершин графа называется нумерация вершин, соответствующая порядку их обхода при поиске в глубину [7,с.35].*

(1) Касьянов В.H., Сабельфельд В.К. Сборник заданий по практикуму на ЭВМ. - М.: Наука, 1986. - 272 с.

(2) Липский В. Комбинаторика для программистов. - М.: Мир, 1988. - 213 с.

(3) Крюков А.П., Радионов А.Я., Таранов А.Ю., Шаблыгин Е.М. Программирование на языке R-Лисп. - М.: Радио и связь, 1991. - 192 с.

(4) Рейнгольд Э., Нивергельт Ю., Део H. Комбинаторные алгоритмы. Теория и практика. - М.: Мир, 1980. - 476 с.

(4) Пападимитриу Х., Стайнглиц К. Комбинаторная оптимизация. Алгоритмы и сложность. - М.: Мир, 1985. - 512 с.

(6) Лекции по теории графов / В.А.Емеличев, О.И.Мельников, В.И.Сарванов, Р.И.Тышкевич - М.: Наука, 1990. - 384 с.

(7) Евстигнеев В.А. Применение теории графов в программировании. - М.: Наука, 1985. - 352 с.

На следующем шаге мы остановимся на ***алгоритме обхода графа в ширину***.

## Обход графов в ширину

На этом шаге мы рассмотрим ***алгоритм обхода графов в ширину***.

Перейдем теперь к другому алгоритму обхода графа, известному под названием ***обход в ширину*** (***поиск в ширину***). Прежде чем описать его, отметим, что при обходе в глубину чем ***позднее*** будет посещена вершина, тем ***раньше*** она будет использована. Это прямое следствие того факта, что просмотренные, но еще не использованные вершины накапливаются в стеке.

Обход графа в ширину, грубо говоря, основывается на ***замене стека очередью***. После такой модификации чем раньше посещается вершина (помещается в очередь), тем раньше она используется (удаляется из очереди). Использование вершины происходит с помощью просмотра сразу всех еще непросмотренных вершин, смежных этой вершины. Таким образом, "поиск ведется как бы во всех возможных направлениях одновременно" [1, с.131]:

**#include** <iostream.h>

**#define** TRUE 1

**#define** FALSE 0

**typedef** **int** Boolean;

**typedef** **struct** Leader \*Lref; **// Тип: указатель на заголовочный узел.**

**typedef** **struct** Trailer \*Tref; **// Тип: указатель на дуговой узел.**

**//Описание типа заголовочного узла.**

**typedef** **struct** Leader

{

**int** Key; **//Имя заголовочного узла.**

**int** Count; **//Количество предшественников.**

Boolean Flag; **//Флаг посещения узла при обходе.**

Tref Trail; **//Указатель на список смежности.**

Lref Next; **//Указатель на следующий узел в списке заголовочных узлов.**

};

**//Описание типа дугового узла.**

**typedef** **struct** Trailer

{

Lref Id;

Tref Next;

};

**//Описание типа узла очереди.**

**typedef** Lref TipElement; **//Указатель на звено заголовочного списка.**

**typedef** **struct** Zveno \*svqz;

**typedef** **struct** Zveno

{

TipElement Element; **//Указатель на список смежности.**

svqz Sled;

};

**class** Spisok

{

**private**:

Lref Head; **//Указатель на голову списка заголовочных узлов.**

Lref Tail; **//Указатель на фиктивный элемент**

**// в конце списка заголовочных узлов.**

**void** Udalenie\_A (svqz \*, svqz \*, TipElement \*);

**void** Dobavlenie (svqz \*, svqz \*, TipElement);

**void** SearchGraph (int, Lref \*);

**public**:

Spisok() {**//Инициализация списка заголовочных узлов.**

Head = Tail = **new** (Leader); }

Lref GetHead() { **return** Head; }

Lref GetTail() { **return** Tail; }

**void** MakeGraph ();

**void** PrintGraph ();

**void** Breadth\_First\_Search (Lref);

};

**void** main ()

{

Spisok A;

Lref t; **//Рабочий указатель для перемещения**

**// по списку заголовочных звеньев.**

**//Построение графа и вывод его структуры Вирта.**

A.MakeGraph ();

A.PrintGraph (); cout<<endl;

**//Нерекурсивный обход графа в ширину.**

cout<<"Результат нерекурсивного обхода...\n";

t = A.GetHead();

**while** (t!=A.GetTail())

{ (\*t).Flag = TRUE; t = (\*t).Next; }

A.Breadth\_First\_Search (A.GetHead()); cout<<endl;

}

**void** Spisok::SearchGraph (**int** w, Lref \*h)

**//Функция возвращает указатель на заголовочный узел**

**//с ключом w в графе, заданном структурой Вирта с указателем Head.**

{

\*h = Head; (\*Tail).Key = w;

**while** ((\*\*h).Key!=w) \*h = (\*\*h).Next;

**if** (\*h==Tail)

**//В списке заголовочных узлов нет узла с ключом w.**

**//Поместим его в конец списка Head.**

{ Tail = **new** (Leader); (\*\*h).Count = 0;

(\*\*h).Trail = **NULL**; (\*\*h).Next = Tail; }

}

**void** Spisok::MakeGraph ()

**//Функция возвращает указатель Head на структуру**

**//Вирта, соответствующую ориентированному графу.**

{

**int** x,y;

Lref p,q; **//Рабочие указатели.**

Tref t,r; **//Рабочие указатели.**

Boolean Res; **//Флаг наличия дуги.**

cout<<"Вводите начальную вершину дуги: ";

cin>>x;

**while** (x!=0)

{

cout<<"Вводите конечную вершину дуги: "; cin>>y;

**//Определим, существует ли в графе дуга (x,y)?**

SearchGraph (x, &p); SearchGraph (y,&q);

r = (\*p).Trail; Res = FALSE;

**while** ((r!=NULL)&&(!Res))

**if** ((\*r).Id==q) Res = TRUE;

**else** r = (\*r).Next;

**if** (!Res) **//Если дуга отсутствует, то поместим её в граф.**

{ t = **new** (Trailer); (\*t).Id = q;

(\*t).Next = (\*p).Trail; (\*p).Trail = t; (\*q).Count++; }

cout<<"Вводите начальную вершину дуги: "; cin>>x;

}

}

**void** Spisok::PrintGraph ()

**//Вывод структуры Вирта, заданной указателем**

**//Head и соответствующей ориентированному графу.**

{

Lref p; **//Рабочий указатель.**

Tref q; **//Рабочий указатель.**

p = Head;

**while** (p!=Tail)

{

cout<<(\*p).Key<<"("; q = (\*p).Trail;

**while** (q!=NULL)

{ cout<<(\*(\*q).Id).Key; q = (\*q).Next; }

cout<<")"; p = (\*p).Next; cout<<" ";

}

}

**void** Spisok::Dobavlenie (svqz \*L, svqz \*R, TipElement elem)

**//Добавление элемента elem в очередь, заданную указателями L и R.**

{

svqz K = **new** (Zveno);

K->Element = elem; K->Sled = **NULL**;

**if** (\*L==NULL)

{ (\*L) = K; (\*R) = K; }

**else** { (\*R)->Sled = K; (\*R) = K; }

}

**void** Spisok::Udalenie\_A (svqz \*L, svqz \*R, TipElement \*A)

**//Удаление элемента из очереди, заданной указателями L и R и**

**//помещение удаленного элемента в переменную A.**

{

svqz q;

**if** ((\*L)!=NULL)

**if** ((\*L)->Sled!=NULL)

{

(\*A) = (\*L)->Element; q = (\*L);

(\*L) = (\*L)->Sled; **delete** q;

}

**else** {

(\*A) = (\*L)->Element; **delete** \*L;

(\*L) = (\*R) = **NULL**;

}

}

**void** Spisok::Breadth\_First\_Search (Lref H)

**//Обход графа в ширину, начиная с вершины, заданной указателем H**

**//(нерекурсивный обход).**

{

svqz L; **//Указатель на начало очереди.**

svqz R; **//Указатель на конец очереди.**

Lref p; **//Рабочий указатель.**

Tref t; **//Рабочий указатель.**

L = R = **NULL**; **//Создание пустой очереди.**

Dobavlenie (&L,&R,H); H->Flag = FALSE;

**while** ( L!=NULL )

**//Пока очередь не пуста...**

{

Udalenie\_A (&L,&R,&p);

cout << p->Key << " "; **//Посещение вершины.**

t = p->Trail;

**while** ( t!=NULL )

{

**if** ( t->Id->Flag )

{

Dobavlenie (&L,&R,t->Id);

t->Id->Flag = FALSE;

}

t = t->Next;

}

}

}

Текст этой программы можно взять [здесь](http://khpi-iip.mipk.kharkiv.edu/library/datastr/book_sod/kgsu/din97_1.zip).

Результат работы приложения изображен на рисунке 1:

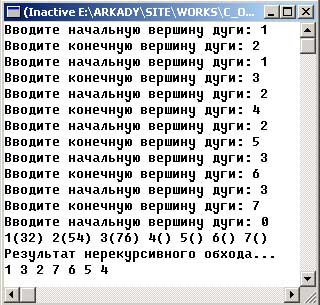


Рис.1. Результат работы приложения

***Замечания.***

1. *Как и в случае обхода графа в глубину, функцию* ***Breadth\_First\_Search()*** *можно использовать без всяких модификаций и тогда, когда мы работаем с* ***ориентированным графом****. Очевидно, что тогда посещаются только те вершины, до которых существует путь от вершины, с которой мы начинаем обход.*
2. *Способ обхода дерева в ширину, называемый иногда* ***обходом в гоpизонтальном поpядке****, основан на посещении вершин деpева слева напpаво, уpовень за уpовнем вниз от коpня. Пpиведенный ниже* ***нерекурсивный*** *алгоpитм позволяет совершить* ***обход деpева в шиpину****, используя две очеpеди* ***O1*** *и* ***O2*** *[2, с.124-125]:*
3. Взять пустые очеpеди **O1** и **O2**.
4. Поместить коpень в очеpедь **O1**.
5. **while** (Одна из очеpедей **O1** и **O2** не пуста)
6. **if** ( **O1** не является пустой )
7. {
8. Пусть **p** - узел, находящийся в голове очеpеди **O1**.
9. Посетить веpшину **p** и удалить ее из **O1**.
10. Поместить всех сыновей веpшины **p** в очеpедь **O2**, начиная
11. со стаpшего сына.
12. }
13. **else** В качестве **O1** взять непустую очеpедь **O2**,
14. а в качестве **O2** взять пустую очеpедь **O1**.

Перейдем к реализации алгоритма ***обхода графа в ширину*** на языке **LISP** [1, с.131].

(DEFUN TESTBF (LAMBDA NIL

(PRINT "Построение графа.")

(SETQ GRAPH NIL) **;Инициализация**

(PRINT "Введите список вершин графа:") (SETQ NODE (READ))

(PRINT "Введите список списков смежных вершин:")

(SETQ NODELIST (READ))

(PRINT (SETQ GRAPH (PAIRLIS NODE NODELIST GRAPH)))

(PRINT "Приступим к обходу графа в ширину...")

(PRINT "Введите вершину графа, с которой начнется обход:")

(SETQ ROOT (READ))

(BREADTHFIRST GRAPH ROOT)

))

**; ------------------------------------**

(DEFUN PAIRLIS (LAMBDA (KEY ADJ GRAPH)

**; Построение графа из списка вершин KEY и списка списков**

**; смежных вершин ADJ путем добавления к существующему**

**; графу GRAPH**

(COND ( (**NULL** KEY) GRAPH )

( (**NULL** ADJ) GRAPH )

( T (CONS (CONS (CAR KEY) (CAR ADJ))

(PAIRLIS (CDR KEY) (CDR ADJ) GRAPH)) )

)

))

**; --------------------------------------**

(DEFUN BREADTHFIRST (LAMBDA (GRAPH ROOT)

**; Обход графа GRAPH в ширину, начиная с вершины ROOT**

**; GRAPH - граф, представленный структурой смежности в**

**; виде ассоциативного списка,**

**; ROOT - вершина графа, с которой начинается обход,**

**; Результат: список вершин графа в порядке посещения**

(BRFI

GRAPH

(LIST ROOT) **; Вершина ROOT уже просмотрена**

(NEIGHBOUR3 ROOT GRAPH) **; Ожидают просмотра вершины, смежные**

**; вершине ROOT**

)

))

**; ---------------------------------------**

(DEFUN BRFI (LAMBDA (GRAPH VISITED QUEUE)

**; GRAPH - граф, представленный структурой смежности в**

**; виде ассоциативного списка,**

**; VISITED - список уже просмотренных вершин**

**; QUEUE - очередь вершин, ожидающих просмотра**

(COND ( (**NULL** QUEUE) (REVERSE VISITED) )

( T (COND ( (MEMBER (CAR QUEUE) VISITED)

**; Первая вершина в списке вершин, ожидающих**

**; просмотра уже посещалась**

(BRFI GRAPH VISITED (CDR QUEUE)) )

( T (BRFI

GRAPH

**; Вершина (CAR QUEUE) посещена**

(CONS (CAR QUEUE) VISITED)

**; В очередь вершин, ожидающих просмотра,**

**; помещаются вершины, смежные посещенной**

(APPEND

QUEUE

(NEIGHBOUR3 (CAR QUEUE) GRAPH)))

)

)

)

)

))

**; ---------------------------------**

(DEFUN NEIGHBOUR3 (LAMBDA (X GRAPH)

**; Функция возвращает список вершин графа GRAPH, смежных с**

**; вершиной X**

(COND ( (**NULL** (ASSOC X GRAPH)) NIL )

( T (CDR (ASSOC X GRAPH)) )

)

))

Текст этой библиотеки можно взять [здесь](http://khpi-iip.mipk.kharkiv.edu/library/datastr/book_sod/kgsu/din97_2.zip).

Тестовые примеры:

1) $ (TESTBF)

Построение графа.

Введите список вершин графа:

(1 2 3 4 5)

Введите список списков смежных вершин:

((2 3 4) (1 3) (1 2 4) (1 3 5) (4))

Приступим к обходу графа в ширину...

Введите вершину графа, с которой начнется обход:

4

(4 1 3 5 2)

2) $ (TESTBF)

Построение графа.

Введите список вершин графа:

(1 2 3 4)

Введите список списков смежных вершин:

((2 3 4) (3) (4) ())

Приступим к обходу графа в ширину...

Введите вершину графа, с которой начнется обход:

2

(2 3 4)

3) $ (TESTBF)

Построение графа.

Введите список вершин графа:

(1 2 3 4 5 6 7)

Введите список списков смежных вершин:

((2 3) (4 5) (6 7) () () () ())

Приступим к обходу графа в ширину...

Введите вершину графа, с которой начнется обход:

1

(1 2 3 4 5 6 7)

Последний тестовый пример иллюстрируется следующим ориентированным графом:

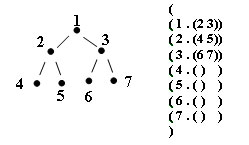


Рис.2. Пример графа

Мы видим, что при ***поиске в ширину*** смежные вершины последней просмотренной вершины добавляются в ***конец***списка **QUEUE** функцией

(APPEND QUEUE (NEIGHBOUR3 (CAR QUEUE) GRAPH))),

в то время как новые вершины для просмотра берутся из ***начала*** этого списка, в который они помещаются функцией

(CONS (CAR QUEUE) VISITED),

т.е. содержимое параметр **QUEUE** используется как ***очередь*** в отличие от параметра **PATH** функции **DEFI**, содержимое которого "работает" как ***стек***. Это приводит к тому, что в процессе работы функции **BRFI** просмотренные вершины в списке **VISITED** расположены в порядке невозрастания их расстояния от начальной вершины **ROOT**.

***Замечания.***

1. *Алгоритмы обхода графа в ширину изложены в монографиях [3, с.36-38], [4, с.92-94], [5, с.198-205], [6, с.238-242].*
2. *Неформально под* ***разметкой*** *в монографии [7, с.28] понимается отображение, сопоставляющее пометки вершинам или дугам графа.*

*Метод разметки используется для построения алгоритмов распознавания различных свойств размеченных графов. Понятие пометки формализует при этом интересующее нас свойство графа или его частей. В качестве анализируемого размеченного графа часто выступает* ***схема программы****. Метод разметки использует тот факт, что свойства вершины графа определяются свойствами некоторых "соседей" этой вершины.*

*Для иллюстрации основной идеи алгоритмов разметки опишем алгоритм такого типа для решения одной очень простой задачи: определения достижимости вершин ориентированного графа от некоторой выделенной его вершины, называемой начальной.*

*Вершина* ***v******достижима от начальной****, если существует путь, ведущий от начальной вершины к вершине* ***v****.*

*Множеством пометок здесь будет служить множество, составленное из двух значений, которые мы назовем "****достижима****" и "****неизвестно****" соответственно. Начальная разметка сопоставляет пометку "****достижима****" начальной вершине графа и "****неизвестно****" всем остальным вершинам. Дальнейший процесс построения необходимой разметки будет состоять в повторном применении правила разметки:*

***если*** некоторая вершина помечена пометкой "достижима"

***то*** пометить пометкой "достижима" всех "наследников" этой вершины.

*Процесс завершается, когда будет достигнута* ***стационарная разметка****, т.е. такая, которая не изменяется никаким применением правила разметки.*

*Отметим две особенности процесса разметки.*

*Во-первых, сразу же возникает вопрос о том, существует ли хотя бы одна стационарная разметка. Иначе говоря, может ли завершиться процесс разметки?*

*Другой вопрос связан с единственностью стационарной разметки. Дело в том, что процесс разметки описан как недетерминированный в том смысле, что на каждом его шаге имеется возможность выбирать, к какой из вершин применять правило разметки. Верно ли, что результирующая стационарная разметка (если она существует) не зависит от того, в каком порядке и к каким вершинам будет применяться правило разметки?*

*Естественно нас интересуют преимущественно такие процессы разметки, когда ответ на оба сформулированных выше вопроса положителен, т.е. когда стационарная разметка существует и единственна. Такие процессы называются* ***алгоритмами разметки****.*

*Ясно, что изученные нами алгоритмы обхода графа представляют собой* ***простейшие алгоритмы разметки****.*

(1) Крюков А.П., Радионов А.Я., Таранов А.Ю., Шаблыгин Е.М. Программирование на языке R-Лисп. - М.: Радио и связь, 1991. - 192 с.

(2) Касьянов В.H., Сабельфельд В.К. Сборник заданий по практикуму на ЭВМ. - М.: Наука, 1986. - 272 с.

(3) Евстигнеев В.А. Применение теории графов в программировании. - М.: Наука, 1985. - 352 с.

(4) Липский В. Комбинаторика для программистов. - М.: Мир, 1988. - 213 с.

(5) Пападимитриу Х., Стайнглиц К. Комбинаторная оптимизация. Алгоритмы и сложность. - М.: Мир, 1985. - 512 с.

(6) Кук Д., Бейз Г. Компьютерная математика. - М.: Наука, 1990. - 384 с.

(7) Котов В.Е., Сабельфельд В.К. Теория схем программ. - М.: Наука, 1991. - 248 с.

На следующем шаге мы рассмотрим ***алгоритмы вычисления пути между фиксированными вершинами***.

## Путь между фиксированными вершинами

На этом шаге мы рассмотрим ***алгоритм нахождения пути между вершинами***.

Оба вида обхода графа - в глубину и в ширину могут быть использованы ***для нахождения пути (цепи) между фиксированными вершинами* u *и* v** [1, с.92]. Достаточно начать обход графа с вершины **v** и вести его до момента посещения вершины **u**.

***Преимуществом обхода графа в глубину*** является тот факт, что в момент посещения вершины u стек содержит последовательность вершин, определяющую путь (цепь) из **v** в **u**. Это становится очевидным, если отметить, что каждая вершина, помещаемая в стек, является смежной вершиной верхнего элемента стека.

Однако ***недостатком*** использования алгоритма обхода графа в глубину для поиска пути между данными вершинами является то, что полученный таким образом путь в общем случае ***не будет кратчайшим путем (кратчайшей цепью)***.

Пример 1. Поиск пути между двумя заданными вершинами в графе с использованием обхода графа в глубину. Граф представлен структурой Вирта.

**#include** <iostream.h>

**#define** TRUE 1

**#define** FALSE 0

**typedef** **int** Boolean;

**typedef** **struct** L \*Lref; **// Тип: указатель на заголовочный узел.**

**typedef** **struct** T \*Tref; **// Тип: указатель на дуговой узел.**

**//Описание типа заголовочного узла.**

**typedef** **struct** L

{

**int** Key; **//Имя заголовочного узла.**

**int** Count; **//Количество предшественников.**

Boolean Flag; **//Флаг посещения узла при обходе.**

Tref Trail; **//Указатель на список смежности.**

Lref Next; **//Указатель на следующий узел в списке заголовочных узлов.**

} Leader;

**//Описание типа дугового узла.**

**typedef** **struct** T

{

Lref Id;

Tref Next;

} Trailer;

**//Описание типа узла стека.**

**typedef** Tref TipElement;

**typedef** **struct** Zveno \*svqz;

**typedef** **struct** Zveno

{

TipElement Element; **//Указатель на список смежности.**

svqz Sled;

} St;

**class** Spisok

{

**private**:

Lref Head; **//Указатель на голову списка заголовочных узлов.**

Lref Tail; **//Указатель на фиктивный элемент**

**// в конце списка заголовочных узлов.**

**void** SearchGraph (int, Lref \*);

**void** W\_S (svqz \*, TipElement);

**void** YDALENIE (svqz \*, TipElement \*);

**public**:

Spisok() {**//Инициализация списка заголовочных узлов.**

Head = Tail = **new** (Leader); }

Lref GetHead() { **return** Head; }

Lref GetTail() { **return** Tail; }

**void** MakeGraph ();

**void** PrintGraph ();

**void** Path\_Depth\_First\_Search (int, int);

};

**void** main ()

{

Spisok A;

**int** B,E;

Lref t; **//Рабочий указатель для перемещения**

**// по списку заголовочных звеньев.**

**//Построение графа и вывод его структуры Вирта.**

A.MakeGraph ();

A.PrintGraph (); cout<<endl;

**//Определение пути между двумя заданными вершинами графа.**

t = A.GetHead();

**while** (t!=A.GetTail())

{ (\*t).Flag = TRUE; t = (\*t).Next; }

cout << "Введите начальную вершину пути: "; cin >> B;

cout << "Введите конечную вершину пути : "; cin >> E;

cout << "Искомый путь: ";

A.Path\_Depth\_First\_Search (B,E); cout<<endl;

}

**void** Spisok::SearchGraph (**int** w, Lref \*h)

**//Функция возвращает указатель на заголовочный узел**

**//с ключом w в графе, заданном структурой Вирта с указателем Head.**

{

\*h = Head; (\*Tail).Key = w;

**while** ((\*\*h).Key!=w) \*h = (\*\*h).Next;

**if** (\*h==Tail)

**//В списке заголовочных узлов нет узла с ключом w.**

**//Поместим его в конец списка Head.**

{ Tail = **new** (Leader); (\*\*h).Count = 0;

(\*\*h).Trail = **NULL**; (\*\*h).Next = Tail; }

}

**void** Spisok::MakeGraph ()

**//Функция возвращает указатель Head на структуру**

**//Вирта, соответствующую ориентированному графу.**

{

**int** x,y;

Lref p,q; **//Рабочие указатели.**

Tref t,r; **//Рабочие указатели.**

Boolean Res; **//Флаг наличия дуги.**

cout<<"Вводите начальную вершину дуги: ";

cin>>x;

**while** (x!=0)

{

cout<<"Вводите конечную вершину дуги: "; cin>>y;

**//Определим, существует ли в графе дуга (x,y)?**

SearchGraph (x, &p); SearchGraph (y,&q);

r = (\*p).Trail; Res = FALSE;

**while** ((r!=**NULL**)&&(!Res))

**if** ((\*r).Id==q) Res = TRUE;

**else** r = (\*r).Next;

**if** (!Res) **//Если дуга отсутствует, то поместим её в граф.**

{ t = **new** (Trailer); (\*t).Id = q;

(\*t).Next = (\*p).Trail; (\*p).Trail = t; (\*q).Count++; }

cout<<"Вводите начальную вершину дуги: "; cin>>x;

}

}

**void** Spisok::PrintGraph ()

**//Вывод структуры Вирта, заданной указателем**

**//Head и соответствующей ориентированному графу.**

{

Lref p; **//Рабочий указатель.**

Tref q; **//Рабочий указатель.**

p = Head;

**while** (p!=Tail)

{

cout<<"("<<(\*p).Key; q = (\*p).Trail;

**while** (q!=**NULL**)

{ cout<<(\*(\*q).Id).Key; q = (\*q).Next; }

cout<<")"; p = (\*p).Next; cout<<" ";

}

}

**void** Spisok::W\_S (svqz \*stk, TipElement el)

**//Помещение элемента el в стек stk.**

{

svqz q=new (St);

(\*q).Element = el; (\*q).Sled = \*stk; \*stk = q;

}

**void** Spisok::YDALENIE (svqz \*stk, TipElement \*klad)

**//Удаление звена из стека, заданного указателем \*stk.**

**//Значение информационного поля удаляемого звена сохраня-**

**//ется в параметре klad.**

{

svqz q;

**if** (\*stk==**NULL**) cout<<"Попытка выбора из пустого стека!\n";

**else**

{ \*klad = (\*\*stk).Element; q = \*stk; \*stk = (\*\*stk).Sled; **delete** q; }

}

**void** Spisok::Path\_Depth\_First\_Search (**int** B, **int** E)

**//Путь между вершинами B и E в графе, заданном указателем Head.**

{

Lref s,r;

Tref t;

svqz UkZv; **//Рабочий указатель для перемещения по стеку.**

svqz Stack = **NULL**; **//Стек пуст.**

**//Определим указатель на вершину B**

s = Head;

**while** (s!=Tail)

{

**if** (s->Key==B) r = s;

s = s->Next;

}

**//Посетим первую непосещенную вершину графа и**

**//поместим ее в первоначально пустой стек.**

**if** (r->Key==E) goto Metka;

r->Flag = FALSE; W\_S (&Stack,r->Trail);

**while** (Stack!=**NULL**)

{

**//Рассмотрим "верхушку" стека.**

t = Stack->Element;

**if** (t->Id->Trail!=**NULL**)

**//У рассматриваемой вершины есть смежные вершины.**

{

**if** (t->Id->Flag)

**//У рассматриваемой вершины есть**

**//непосещенные смежные вершины.**

{

**//Посетим рассматриваемую вершину**

**//и поместим указатель на ее список смежности в стек.**

**if** (t->Id->Key==E) goto Metka;

t->Id->Flag = FALSE;

W\_S (&Stack,t->Id->Trail);

}

**//У рассматриваемой вершины нет**

**//непосещенных смежных вершин.**

**else**

{

t = Stack->Element;

**if** (t->Next!=**NULL**)

**//Заменяем верхушку стека указателем**

**//на следующий элемент списка смежности...**

{

YDALENIE (&Stack,&t);

W\_S (&Stack,t->Next);

}

**//или удаляем верхушку стека.**

**else** YDALENIE (&Stack,&t);

}

}

**//У рассматриваемой вершины нет смежных вершин.**

**else**

{

**if** (t->Id->Flag)

**//Посетим рассматриваемую вершину.**

{

**if** (t->Id->Key==E) goto Metka;

t->Id->Flag = FALSE;

}

t = Stack->Element;

**if** (t->Next!=**NULL**)

**//Заменяем верхушку стека указателем**

**//на следующий элемент списка смежности...**

{

YDALENIE (&Stack,&t);

W\_S (&Stack,t->Next);

}

**//или удаляем верхушку стека.**

**else** YDALENIE (&Stack,&t);

}

}

Metka:

UkZv = Stack;

**while** ( UkZv!=**NULL** )

{

cout << UkZv->Element->Id->Key << " ";

UkZv = UkZv->Sled;

}

cout << B << endl;

}

Текст этой программы можно взять [здесь](http://khpi-iip.mipk.kharkiv.edu/library/datastr/book_sod/kgsu/din98_1.zip).

Для ориентированного графа, представленного на рисунке 1:

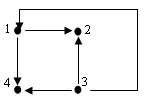


Рис.1. Пример графа

результат работы программы изображен на рисунке 2. Обратите внимание, что найден ***не оптимальный путь!***

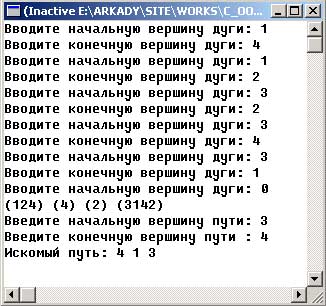


Рис.2. Результат работы приложения

Использование алгоритма обхода графа в ***ширину*** позволяет получить ***кратчайший путь*** между двумя фиксированными вершинами [1, с.93].

Пример 2. Поиск пути между двумя заданными вершинами в графе с использованием обхода графа в ширину (нерекурсивный алгоритм). Граф представлен структурой Вирта.

**#include** <iostream.h>

**#define** TRUE 1

**#define** FALSE 0

**typedef** **int** Boolean;

**typedef** **struct** Leader \*Lref; **// Тип: указатель на заголовочный узел.**

**typedef** **struct** Trailer \*Tref; **// Тип: указатель на дуговой узел.**

**//Описание типа заголовочного узла.**

**typedef** **struct** Leader

{

**int** Key; **//Имя заголовочного узла.**

**int** Count; **//Количество предшественников.**

Boolean Flag; **//Флаг посещения узла при обходе.**

Tref Trail; **//Указатель на список смежности.**

Lref Next; **//Указатель на следующий узел в списке заголовочных узлов.**

};

**//Описание типа дугового узла.**

**typedef** **struct** Trailer

{

Lref Id;

Tref Next;

};

**//Описание типа узла очереди.**

**typedef** Lref TipElement; **//Указатель на звено заголовочного списка.**

**typedef** **struct** Zveno \*svqz;

**typedef** **struct** Zveno

{

TipElement Element; **//Указатель на список смежности.**

svqz Sled;

};

**class** Spisok

{

**private**:

Lref Head; **//Указатель на голову списка заголовочных узлов.**

Lref Tail; **//Указатель на фиктивный элемент**

**// в конце списка заголовочных узлов.**

**void** Udalenie\_A (svqz \*, svqz \*, TipElement \*);

**void** Dobavlenie (svqz \*, svqz \*, TipElement);

**void** SearchGraph (int, Lref \*);

**public**:

Spisok() {**//Инициализация списка заголовочных узлов.**

Head = Tail = **new** (Leader); }

Lref GetHead() { **return** Head; }

Lref GetTail() { **return** Tail; }

**void** MakeGraph ();

**void** PrintGraph ();

**void** Path\_Breadth\_First\_Search (Lref, int, int);

};

**void** main ()

{

Spisok A;

**int** B,E;

Lref t; **//Рабочий указатель для перемещения**

**// по списку заголовочных звеньев.**

**//Построение графа и вывод его структуры Вирта.**

A.MakeGraph ();

A.PrintGraph (); cout<<endl;

**//Определение пути между двумя заданными вершинами графа.**

t = A.GetHead();

**while** (t!=A.GetTail())

{ (\*t).Flag = TRUE; t = (\*t).Next; }

cout << "Введите начальную вершину пути: "; cin >> B;

cout << "Введите конечную вершину пути : "; cin >> E;

cout << "Искомый путь: ";

A.Path\_Breadth\_First\_Search(A.GetHead(),B,E); cout<<endl;

}

**void** Spisok::SearchGraph (**int** w, Lref \*h)

**//Функция возвращает указатель на заголовочный узел**

**//с ключом w в графе, заданном структурой Вирта с указателем Head.**

{

\*h = Head; (\*Tail).Key = w;

**while** ((\*\*h).Key!=w) \*h = (\*\*h).Next;

**if** (\*h==Tail)

**//В списке заголовочных узлов нет узла с ключом w.**

**//Поместим его в конец списка Head.**

{ Tail = **new** (Leader); (\*\*h).Count = 0;

(\*\*h).Trail = **NULL**; (\*\*h).Next = Tail; }

}

**void** Spisok::MakeGraph ()

**//Функция возвращает указатель Head на структуру**

**//Вирта, соответствующую ориентированному графу.**

{

**int** x,y;

Lref p,q; **//Рабочие указатели.**

Tref t,r; **//Рабочие указатели.**

Boolean Res; **//Флаг наличия дуги.**

cout<<"Вводите начальную вершину дуги: ";

cin>>x;

**while** (x!=0)

{

cout<<"Вводите конечную вершину дуги: "; cin>>y;

**//Определим, существует ли в графе дуга (x,y)?**

SearchGraph (x, &p); SearchGraph (y,&q);

r = (\*p).Trail; Res = FALSE;

**while** ((r!=**NULL**)&&(!Res))

**if** ((\*r).Id==q) Res = TRUE;

**else** r = (\*r).Next;

**if** (!Res) **//Если дуга отсутствует, то поместим её в граф.**

{ t = **new** (Trailer); (\*t).Id = q;

(\*t).Next = (\*p).Trail; (\*p).Trail = t; (\*q).Count++; }

cout<<"Вводите начальную вершину дуги: "; cin>>x;

}

}

**void** Spisok::PrintGraph ()

**//Вывод структуры Вирта, заданной указателем**

**//Head и соответствующей ориентированному графу.**

{

Lref p; **//Рабочий указатель.**

Tref q; **//Рабочий указатель.**

p = Head;

**while** (p!=Tail)

{

cout<<(\*p).Key<<"("; q = (\*p).Trail;

**while** (q!=**NULL**)

{ cout<<(\*(\*q).Id).Key; q = (\*q).Next; }

cout<<")"; p = (\*p).Next; cout<<" ";

}

}

**void** Spisok::Dobavlenie (svqz \*L, svqz \*R, TipElement elem)

**//Добавление элемента elem в очередь, заданную указателями L и R.**

{

svqz K = **new** (Zveno);

K->Element = elem; K->Sled = **NULL**;

**if** (\*L==**NULL**)

{ (\*L) = K; (\*R) = K; }

**else** { (\*R)->Sled = K; (\*R) = K; }

}

**void** Spisok::Udalenie\_A (svqz \*L, svqz \*R, TipElement \*A)

**//Удаление элемента из очереди, заданной указателями L и R и**

**//помещение удаленного элемента в переменную A.**

{

svqz q;

**if** ((\*L)!=**NULL**)

**if** ((\*L)->Sled!=**NULL**)

{

(\*A) = (\*L)->Element; q = (\*L);

(\*L) = (\*L)->Sled; **delete** q;

}

**else** {

(\*A) = (\*L)->Element; **delete** \*L;

(\*L) = (\*R) = **NULL**;

}

}

**void** Spisok::Path\_Breadth\_First\_Search (Lref H, **int** B, **int** E)

**//Путь в графе, заданном указателем H, между вершинами B и E.**

{

svqz L; **//Указатель на начало очеpеди.**

svqz R; **//Указатель на конец очеpеди.**

Lref p; **//Рабочий указатель.**

Tref t; **//Рабочий указатель.**

**int** Pred[30]; **//Элемент Pred[i] содержит вершину графа,**

**//"предшествующую" данной.**

**int** i;

L = R = **NULL**; **// Построение пустой очеpеди.**

**//Определим указатель на вершину B и поместим его в очередь.**

p = H;

**while** ( p!=Tail )

{

**if** ( p->Key==B )

{

Dobavlenie (&L,&R,p);

p->Flag = FALSE;

}

p = p->Next;

}

**//Пока очеpедь не пуста...**

**while** (L!=**NULL**)

{

Udalenie\_A (&L,&R,&p);

t = p->Trail;

**while** (t!=**NULL**)

{

**if** (t->Id->Flag)

{

Dobavlenie (&L,&R,t->Id);

t->Id->Flag = FALSE;

Pred [t->Id->Key] = p->Key;

}

t = t->Next;

}

}

i = E;

cout << E << ' ';

**while** (i!=B)

{ cout << Pred[i] <<' '; i = Pred[i]; }

cout << endl;

}

Текст этой программы можно взять [здесь](http://khpi-iip.mipk.kharkiv.edu/library/datastr/book_sod/kgsu/din98_2.zip).

Результат работы программы для ориентированного графа, изображенного на рисунке 1, отражен на рисунке 3:

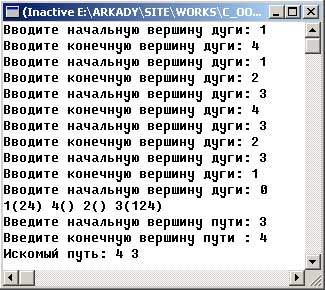


Рис.3. Результат работы приложения

Вернемся теперь к ***функциональному программированию***.

Мы уже упоминали о том, что параметр **PATH** функции **DEFI** (смотри [шаг 96](http://khpi-iip.mipk.kharkiv.edu/library/datastr/book_sod/kgsu/din_0096.html#1)) описывает ***путь из начальной вершины в просматриваемую***. Поэтому алгоритм обхода графа в глубину можно легко модифицировать в ***алгоритм поиска пути между заданными вершинами*** (например, вершинами **ROOT** и **END**).

Пример 3.

(DEFUN TESTWAY (LAMBDA NIL

(PRINT "Построение графа.")

(SETQ GRAPH NIL) **;Инициализация**

(PRINT "Введите список вершин графа:") (SETQ NODE (READ))

(PRINT "Введите список списков смежных вершин:")

(SETQ NODELIST (READ))

(PRINT (SETQ GRAPH (PAIRLIS NODE NODELIST GRAPH)))

(PRINT "Приступим к поиску пути между заданными вершинами")

(PRINT " с использованием обхода графа в глубину... ")

(PRINT "Введите начальную вершину пути:") (SETQ ROOT (READ))

(PRINT "Введите конечную вершину пути:") (SETQ END (READ))

(WAY GRAPH ROOT END)

))

**; ------------------------------------**

(DEFUN PAIRLIS (LAMBDA (KEY ADJ GRAPH)

**; Построение графа из списка вершин KEY и списка списков**

**; смежных вершин ADJ путем добавления к существующему**

**; графу GRAPH**

(COND ( (**NULL** KEY) GRAPH )

( (**NULL** ADJ) GRAPH )

( T (CONS (CONS (CAR KEY) (CAR ADJ))

(PAIRLIS (CDR KEY) (CDR ADJ) GRAPH)) )

)

))

**; ---------------------------------**

(DEFUN WAY (LAMBDA (GRAPH ROOT END)

**; Построение пути в графе GRAPH между вершинами ROOT и END**

(COND ( (**NULL** GRAPH) NIL )

( T (DEFI GRAPH (LIST ROOT) (LIST ROOT) END) )

)

))

**; ------------------------------------------**

(DEFUN DEFI (LAMBDA (GRAPH VISITED PATH END)

**; GRAPH - граф, представленный структурой смежности в**

**; виде ассоциативного списка,**

**; VISITED - список уже посещенных вершин,**

**; PATH - список вершин, определяющий путь посещения**

**; END - конечная вершина пути**

(COND ( (**NULL** PATH) (REVERSE VISITED) )

( T (COND ( (**NULL** (EXPND GRAPH VISITED (CAR PATH)))

(DEFI GRAPH VISITED (CDR PATH) END)

)

( (EQ (EXPND GRAPH VISITED (CAR PATH)) END)

(REVERSE (CONS END PATH)) )

( T (DEFI GRAPH

(CONS (EXPND GRAPH VISITED

(CAR PATH))

VISITED)

(CONS (EXPND GRAPH VISITED

(CAR PATH))

PATH)

END) )) )

)

))

**; -----------------------------------------**

(DEFUN EXPND (LAMBDA (GRAPH VISITED VERTEX)

**; Выбор в графе GRAPH следующей еще не просмотренной**

**; вершины, смежной с вершиной 2VERTEX**

(COND ( (**NULL** (NEIGHBOUR3 VERTEX GRAPH)) NIL )

( T (FIRSTNOTVISITED

VISITED

(NEIGHBOUR3 VERTEX GRAPH))

)

)

))

**; --------------------------------------------**

(DEFUN FIRSTNOTVISITED (LAMBDA (VISITED VLIST)

**; Поиск первой непосещенной вершины в списке VLIST**

**; VISITED - список уже посещенных вершин**

(COND ( (**NULL** VLIST) NIL )

( T (COND ( (**NULL** (MEMBER (CAR VLIST) VISITED))

(CAR VLIST)

)

( T (FIRSTNOTVISITED

VISITED

(CDR VLIST))

)

)

)

)

))

**; ---------------------------------**

(DEFUN NEIGHBOUR3 (LAMBDA (X GRAPH)

**; Функция возвращает список вершин графа GRAPH, смежных с**

**; вершиной X**

(COND ( (**NULL** (ASSOC X GRAPH)) NIL )

( T (CDR (ASSOC X GRAPH)) )

)

))

Текст этой библиотеки можно взять [здесь](http://khpi-iip.mipk.kharkiv.edu/library/datastr/book_sod/kgsu/din98_3.zip).

Рассмотрим тестовые примеры для изображенного на рисунке ориентированного графа:

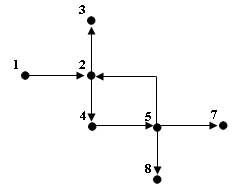


Рис.4. Пример графа

1) $ (TESTWAY)

Построение графа.

Введите список вершин графа:

(1 2 3 4 5 7 8)

Введите список списков смежных вершин:

((2) (3 4) () (5) (2 7 8) () ())

((1 2) (2 3 4) (3) (4 5) (5 2 7 8) (7) (8))

Приступим к поиску пути между заданными вершинами

с использованием обхода графа в глубину...

Введите начальную вершину пути:

1

Введите конечную вершину пути:

5

(1 2 4 5)

2) $ (TESTWAY)

Построение графа.

Введите список вершин графа:

(1 2 3 4 5 7 8)

Введите список списков смежных вершин:

((2) (3 4) () (5) (2 7 8) () ())

((1 2) (2 3 4) (3) (4 5) (5 2 7 8) (7) (8))

Приступим к поиску пути между заданными вершинами

с использованием обхода графа в глубину...

Введите начальную вершину пути:

1

Введите конечную вершину пути:

1

(1 2 3 4 5 7 8)

Обратите внимание на тот факт, что если пути из вершины **ROOT** в вершину **END** не существует, то функция **(WAY GRAPH ROOT END)** ***возвращает результат обхода графа в глубину***.

(1) Липский В. Комбинаторика для программистов. - М.: Мир, 1988. - 213 с.

На следующем шаге мы познакомимся с ***Эйлеровыми путями и циклами***.

## Эйлеровы пути и циклы

На этом шаге мы рассмотрим ***Эйлеровы пути и циклы***.

***Определение 1 [1, с.106]***.

***Эйлеровым путем*** в графе называется путь, проходящий через каждое ребро графа только один раз, т.е. путь ***v1,v2..., vm+1***, такой, что каждое ребро **e**, принадлежащее **E**, появляется в последовательности **v1, v2..., vm+1** в точности один раз как **e={vi,vi+1}**. 0 Иногда граф, обладающий эйлеровым путем, называют ***полуэйлеровым*** [2, с.43].

Если **v1=vm+1**, то такой путь называется ***эйлеровым циклом***, а граф, обладающий эйлеровым циклом, называют ***эйлеровым графом*** [2, с.43].

Задача существования эйлерова пути в заданном графе была решена Л.Эйлером в 1736 г., и представленное им необходимое и достаточное условие существования такого пути считается первой в истории теоремой теории графов.

***Теорема [1, с.106].***

Эйлеров путь в графе существует ***тогда и только тогда***, когда:

* граф связный и
* содержит не более чем две вершины нечетной степени.

Приведем формулировку этой же теоремы в другой терминологии.

***Теорема [2, с.43].***

Связный граф является полуэйлеровым ***тогда и только тогда***, когда в нем не более двух вершин имеют нечетные степени.

Если в связном графе ***нет вершин нечетной степени***, то ***каждый эйлеров путь является циклом***, так как концы эйлерова пути, не являющегося циклом, всегда вершины нечетной степени.

Это утверждение в монографии [2, с.4 24 0] оформлено в виде теоремы.

***Теорема***.

Связный граф G является эйлеровым (***обладает эйлеровым циклом***) ***тогда и только тогда***, когда каждая вершина в **G** имеет четную степень.

***Определение 2 [3, с.182].***

***Эйлеров контур*** 0 в ориентированном графе - это контур, содержащий все дуги и только по одному разу.

Каковы же те графы, которые обладают эйлеровыми контурами?

Граф называется ***псевдосимметрическим***, если в каждой его вершине число исходящих дуг равно числу заходящих. Эта терминология оправдывается тем, что всякий симметрический граф является в то же время псевдосимметрическим.

***Теорема [3, с.182]***.

Граф обладает эйлеровым контуром тогда и только тогда, когда он является связным и псевдосимметрическим.

Реализуем приведенный в [1,с.107] алгоритм нахождения эйлерова цикла в связном графе ***без вершин нечетной степени***, представленном структурой Вирта.

**void** Spisok::Euler ()

**//Построение эйлерова цикла в связном графе без вершин нечетной**

**//степени, заданном структурой Вирта. Начало цикла определено**

**//указателем Head.**

{

Lref v,u; **//Рабочие переменные.**

svqz q; **//Рабочая переменная.**

svqz Stack =**NULL**, CE = **NULL**;

W\_S (&Stack,Head);

**while** (Stack!=**NULL**)

{

**//Взглянем на верхний элемент стека Stack...**

v = Stack->Element;

**if** (v->Trail!=**NULL**)

{

u = v->Trail->Id;

W\_S (&Stack,u);

DeleteGraph (v->Key,u->Key);

DeleteGraph (u->Key,v->Key);

}

**else**

{

YDALENIE (&Stack,&v);

W\_S (&CE,v);

}

}

**//В стеке CE хранится эйлеров цикл.**

q = CE;

**while** (q!=**NULL**)

{ cout << q->Element->Key << ' '; q = q->Sled; }

}

Оценим теперь вычислительную сложность алгоритма [1, с.108]. Для этого отметим, что каждая итерация главного цикла либо помещает вершину в стек **Stack** и удаляет ребро из графа, либо переносит вершину из стека **Stack** в стек **CE**. Таким образом, число итераций этого цикла - **O(m)**, где **m** - количество ребер.

Ухудшить алгоритм может лишь процедура удаления ребра из графа, которая у нас называется DeleteGraph (), и встречается почти в каждой итерации. Она это и делает, ибо уже на поиски удаляемого ребра затрачивается время **2\*O(n)**, где **n** - количество вершин графа.

Из приведенных рассуждений можно сделать вывод, что общая сложность алгоритма есть **O(m\*n)**.

Пример.

**#include** <iostream.h>

**#define** TRUE 1

**#define** FALSE 0

**typedef** **int** Boolean;

**typedef** **struct** L \*Lref; **// Тип: указатель на заголовочный узел.**

**typedef** **struct** T \*Tref; **// Тип: указатель на дуговой узел.**

**//Описание типа заголовочного узла.**

**typedef** **struct** L

{

**int** Key; **//Имя заголовочного узла.**

**int** Count; **//Количество предшественников.**

Boolean Flag; **//Флаг посещения узла при обходе.**

Tref Trail; **//Указатель на список смежности.**

Lref Next; **//Указатель на следующий узел в списке заголовочных узлов.**

} Leader;

**//Описание типа дугового узла.**

**typedef** **struct** T

{

Lref Id;

Tref Next;

} Trailer;

**//Описание типа узла стека.**

**typedef** Lref TipElement;

**typedef** **struct** Zveno \*svqz;

**typedef** **struct** Zveno

{

TipElement Element; **//Указатель на список смежности.**

svqz Sled;

} St;

**class** Spisok

{

**private**:

Lref Head; **//Указатель на голову списка заголовочных узлов.**

Lref Tail; **//Указатель на фиктивный элемент**

**// в конце списка заголовочных узлов.**

**void** SearchGraph (int, Lref \*);

**void** W\_S (svqz \*, TipElement);

**void** YDALENIE (svqz \*, TipElement \*);

Lref Search (**int**);

**void** DeleteGraph (int, int);

**public**:

Spisok() {**//Инициализация списка заголовочных узлов.**

Head = Tail = **new** (Leader); }

**void** MakeGraph ();

**void** PrintGraph ();

**void** Euler ();

};

**void** main ()

{

Spisok A;

**//Построение графа и вывод его структуры Вирта.**

A.MakeGraph ();

A.PrintGraph (); cout<<endl;

**//Построение Эйлерова цикла.**

A.Euler();

A.PrintGraph (); cout<<endl;

}

**void** Spisok::SearchGraph (**int** w, Lref \*h)

**//Функция возвращает указатель на заголовочный узел**

**//с ключом w в графе, заданном структурой Вирта с указателем Head.**

{

\*h = Head; (\*Tail).Key = w;

**while** ((\*\*h).Key!=w) \*h = (\*\*h).Next;

**if** (\*h==Tail)

**//В списке заголовочных узлов нет узла с ключом w.**

**//Поместим его в конец списка Head.**

{ Tail = **new** (Leader); (\*\*h).Count = 0;

(\*\*h).Trail = **NULL**; (\*\*h).Next = Tail; }

}

**void** Spisok::MakeGraph ()

**//Функция возвращает указатель Head на структуру**

**//Вирта, соответствующую ориентированному графу.**

{

**int** x,y;

Lref p,q; **//Рабочие указатели.**

Tref t,r; **//Рабочие указатели.**

Boolean Res; **//Флаг наличия дуги.**

cout<<"Вводите начальную вершину дуги: ";

cin>>x;

**while** (x!=0)

{

cout<<"Вводите конечную вершину дуги: "; cin>>y;

**//Определим, существует ли в графе дуга (x,y)?**

SearchGraph (x, &p); SearchGraph (y,&q);

r = (\*p).Trail; Res = FALSE;

**while** ((r!=**NULL**)&&(!Res))

**if** ((\*r).Id==q) Res = TRUE;

**else** r = (\*r).Next;

**if** (!Res) **//Если дуга отсутствует, то поместим её в граф.**

{ t = **new** (Trailer); (\*t).Id = q;

(\*t).Next = (\*p).Trail; (\*p).Trail = t; (\*q).Count++; }

cout<<"Вводите начальную вершину дуги: "; cin>>x;

}

}

**void** Spisok::PrintGraph ()

**//Вывод структуры Вирта, заданной указателем**

**//Head и соответствующей ориентированному графу.**

{

Lref p; **//Рабочий указатель.**

Tref q; **//Рабочий указатель.**

p = Head;

**while** (p!=Tail)

{

cout<<"("<<(\*p).Key; q = (\*p).Trail;

**while** (q!=**NULL**)

{ cout<<(\*(\*q).Id).Key; q = (\*q).Next; }

cout<<")"; p = (\*p).Next; cout<<" ";

}

}

**void** Spisok::W\_S (svqz \*stk, TipElement el)

**//Помещение элемента el в стек stk.**

{

svqz q=new (St);

(\*q).Element = el; (\*q).Sled = \*stk; \*stk = q;

}

**void** Spisok::YDALENIE (svqz \*stk, TipElement \*klad)

**//Удаление звена из стека, заданного указателем \*stk.**

**//Значение информационного поля удаляемого звена сохраня-**

**//ется в параметре klad.**

{

svqz q;

**if** (\*stk==**NULL**) cout<<"Попытка выбора из пустого стека!\n";

**else**

{ \*klad = (\*\*stk).Element; q = \*stk; \*stk = (\*\*stk).Sled; **delete** q; }

}

**void** Spisok::DeleteGraph (**int** x,int y)

**//Функция возвращает указатель \*Head на структуру**

**//Вирта, соответствующую ориентированному графу и**

**//полученную удалением дуги (x,y).**

{

Lref p,q; **//Рабочие указатели.**

Tref t,r; **//Рабочие указатели.**

Boolean Res; **//Флаг наличия дуги.**

**// Определим, существует ли в графе дуга (x,y)?**

p = Search (x);q = Search (y);

**if** ((p!=**NULL**) && (q!=**NULL**))

{ **// Вершины в графе есть.**

r = (\*p).Trail; Res = FALSE;

**while** ((r!=**NULL**) && (!Res))

**if** ((\*r).Id==q) Res = TRUE;

**else** r = (\*r).Next;

**if** (Res) **//Если дуга существует, то удалим её.**

**if** (r==(\*p).Trail)

{ (\*p).Trail = (\*(\*p).Trail).Next; **delete** r; (\*q).Count--; }

**else** {

t = (\*p).Trail;

**while** ((\*t).Next!=r) t = (\*t).Next;

(\*t).Next = (\*(\*t).Next).Next; **delete** r; (\*q).Count--; }

}

}

Lref Spisok::Search (**int** w)

**//Функция возвращает указатель на заголовочный узел с**

**//ключом w. Если узел отсутствует, то функция возвращает**

**//NULL.**

{

Lref h;

h = Head;

(\*Tail).Key = w; **//Поиск "с барьером".**

**while** ((\*h).Key!=w) h = (\*h).Next;

**if** (h==Tail) **//В списке заголовочных узлов нет узла с ключом w.**

h = **NULL**;

**return** h;

}

**void** Spisok::Euler ()

**//Построение эйлерова цикла в связном графе без вершин нечетной**

**//степени, заданном структурой Вирта. Начало цикла определено**

**//указателем Head.**

{

Lref v,u; **//Рабочие переменные.**

svqz q; **//Рабочая переменная.**

svqz Stack =**NULL**, CE = **NULL**;

W\_S (&Stack,Head);

**while** (Stack!=**NULL**)

{

**//Взглянем на верхний элемент стека Stack...**

v = Stack->Element;

**if** (v->Trail!=**NULL**)

{

u = v->Trail->Id;

W\_S (&Stack,u);

DeleteGraph (v->Key,u->Key);

DeleteGraph (u->Key,v->Key);

}

**else**

{

YDALENIE (&Stack,&v);

W\_S (&CE,v);

}

}

**//В стеке CE хранится эйлеров цикл.**

q = CE;

**while** (q!=**NULL**)

{ cout << q->Element->Key << ' '; q = q->Sled; }

}

Текст этой программы можно взять [здесь](http://khpi-iip.mipk.kharkiv.edu/library/datastr/book_sod/kgsu/din99_1.zip).

***Замечания.***

1. *Приведем алгоритм построения эйлеровой цепи в данном эйлеровом графе. Этот метод известен под названием* ***алгоритма Флёри****.*

***Теорема [2, с.45-46]****.*

*Пусть* ***G*** *- эйлеров граф; тогда следующая процедура всегда возможна и приводит к построению эйлеровой цепи графа* ***G****. Выходя из произвольной вершины u, идем по ребрам графа произвольным образом, соблюдая лишь следующие правила:*

* + *стираем ребра по мере их прохождения и стираем также изолированные вершины, которые при этом образуются;*
  + *на каждом этапе идем по* ***мосту*** *только тогда, когда нет других возможностей (или другими словами, не проходить по ребру, если удаление этого ребра приводит к разбиению графа на две связные компоненты).*

1. *В монографии [4, с.173-175] изложен "изящный и практически эффективный* ***алгоритм Хоанг Туя*** *(1964 г.). При этом, правда, предполагается, что для нахождения какой-нибудь цепи между двумя заданными вершинами, а значит, и цикла, содержащего заданное цикловое ребро, достаточно хорошие алгоритмы известны.*
2. *Ребрам графа* ***G*** *приписаны положительные веса. Требуется найти цикл, проходящий через каждое ребро графа* ***G******по крайней мере один раз*** *и такой, что для него общий вес (а именно сумма величин* ***nj\*c(aj)****, где числа* ***nj*** *показывают, сколько раз проходилось ребро* ***aj****, а* ***c(aj)*** *- вес ребра) минимален.*

*Очевидно, что если* ***G*** *содержит эйлеров цикл, то любой такой цикл будет оптимальным, так как каждое ребро проходится только один раз и вес этого цикла равен тогда сумме* ***c(aj)****, где* ***j*** *изменяется от 1 до* ***m****.*

*Сформулированная задача называется* ***задачей китайского почтальона*** *[5, с.230-231].*

1. *Кёнигсберг (теперь Калининград) расположен на обоих берегах реки Преголя и на двух островах этой реки. Берега реки и два острова соединены семью мостами. Вопрос - поставленный в 1736 г. - состоит в том, можно ли, начав с некоторой точки, совершить прогулку и вернуться в исходную точку, пройдя по каждому мосту ровно один раз. Если каждый берег реки и острова считать вершиной графа, а каждый мост - ребром, то карту можно представить в виде графа, а ответ на поставленный вопрос зависит теперь от существования эйлерова цикла в этом графе.*

*Отрицательное решение Л.Эйлером (Commentarii Acad.Petropoletanae 8(1736),128-140) этой задачи, по-видимому, можно считать первой печатной научной работой по несуществовавшей тогда теории графов.*

*Типичной задачей по эйлеровым графам является задача с такой постановкой: можно ли нарисовать какую-нибудь диаграмму (соединив данные точки линией), не отрывая карандаша от бумаги и не проходя никакую линию дважды.*

1. *[6, с.39]. Эйлеровым графом может быть представлен, например, план выставки; это позволяет так расставить указатели маршрута, чтобы посетитель смог пройти по каждому залу в точности по одному разу.*

(1) Липский В. Комбинаторика для программистов. - М.: Мир, 1988. - 213 с.

(2) Уилсон Р. Введение в теорию графов. - М.: Мир, 1977. - 207 с.

(3) Берж К. Теория графов и ее применения. - М.: ИЛ,1962. - 320 с.

(4) Зыков А.А. Основы теории графов. М.: Наука, 1987. - 384 с.

(5) Кристофидес Н. Теория графов. Алгоритмический подход. - М.: Мир, 1978. - 432 с.

(6) Березина Л.Ю. Графы и их применение. - М.: Просвещение, 1979. - 143 с.

Со следующего шага мы начнем рассматривать ***алгоритмы на графах***.

## Алгоритмы на графах. Кратчайшие пути между всеми парами вершин. Алгоритм Уоршалла

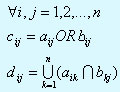
На этом шаге мы рассмотрим ***алгоритм Уоршалла***.

Метод определения матрицы достижимости **P** графа сначала путем вычисления **A, A2, ..., An**, а затем **Bn** является очень громоздким.

Опишем более эффективный метод, основанный на аналогичных идеях. Заметим, что нас не интересует число путей любой конкретной длины из вершины vi в вершину vj. Эта информация, получаемая в процессе вычисления степеней **A**, далее игнорируется. Для того, чтобы сократить объем вычислений, можно отказаться от получения указанной информации и использовать в вычислениях просто реализуемые ***булевские матричные операции***, которые мы определим согласно [1, с.440].

Обозначим ***булевскую сумму* C** двух матриц **A** и **B** размера **n\*n** как http://khpi-iip.mipk.kharkiv.edu/library/datastr/book_sod/kgsu/ris100_1.jpg, а ***булевское произведение* D** - http://khpi-iip.mipk.kharkiv.edu/library/datastr/book_sod/kgsu/ris100_2.jpg.

Элементы матриц **C** и **D** задаются соотношениями



Заметим, что элемент **dij** легко получается путем просмотра **i**-й строки матрицы **A** слева направо и одновременно **j**-го столбца матрицы **B** сверху вниз. Если **k**-й элемент в строке матрицы **A** и **k**-й элемент в столбце матрицы **B** равны 1 для какого-нибудь **k**, то **dij=1**. В противном случае **dij=0**.

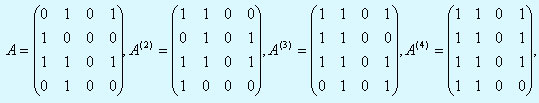
Булевы матрицы более экономичны в вычислительном отношении, чем целочисленные. Действительно, запоминание булевой матрицы требует ***меньшего объема*** оперативной памяти ЭВМ ***по сравнению с целочисленной матрицей той же размерности***. Кроме того, выполнение на компьютере логических операций над булевыми матрицами требует меньшего объема вычислений, чем над целочисленными матрицами тех же размерностей.

Матрица смежностей, так же как и путевая матрица, является булевской матрицей. Заметим, что http://khpi-iip.mipk.kharkiv.edu/library/datastr/book_sod/kgsu/ris100_4.jpg.

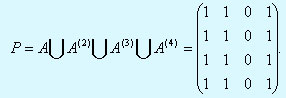
Единственная разница между **A2** и **A(2)** заключается в том, что **A(2)** является булевской матрицей и элемент на пересечении **i**-й строки и **j**-го столбца **A(2)** равен 1 в том случае, когда существует по крайней мере один путь длины 2 из **vi** в **vj**. Аналогичное положение имеет место для **A3** и **A(3)** и в общем случае для **Ar** и **A(r)** при любом целом положительном **r**. Из этих рассуждений ясно, что матрица достижимости **P** задается выражением

http://khpi-iip.mipk.kharkiv.edu/library/datastr/book_sod/kgsu/ris100_5.jpg

Например, если



то



Данный метод получения матрицы достижимости ориентированного графа называется ***алгоритмом Уоршалла***(**Warshall S.A. A Theorem on Boolean Matrices. - J.ACM, 1962, 9, pp.11-12**). Он может быть легко реализован на языке C++.

Программа 1 [2, с.329]. ***Вычисление матрицы достижимости по заданной матрице смежностей с помощью алгоритма Уоршалла***.

**#include** <iostream.h>

**#define** MaxNodes 4

**class** Warshall

{

**private**:

**unsigned** Adj[MaxNodes][MaxNodes]; **//Матрица смежностей.**

**unsigned** Path[MaxNodes][MaxNodes]; **//Матрица достижимости.**

**public**:

**void** Vvod();

**void** TransClose();

**void** Vyvod();

};

**void** Warshall::Vvod()

{

**//Ввод матрицы смежностей заданного графа.**

cout <<"Вводите элементы матрицы смежностей по строкам:\n";

**for** (**int** i=0;i<MaxNodes;i++)

**for** (**int** j=0;j<MaxNodes;j++)

{

cout <<"Введите Adj["<< (i+1) << "]["<< (j+1) << "]: ";

cin >> Adj[i][j];

}

}

**void** Warshall::TransClose()

**//Вычисление матрицы достижимости.**

{

**//Инициализация матрицы Path матрицей смежностей Adj.**

**for** (**int** i=0;i<MaxNodes;i++)

**for** (**int** j=0;j<MaxNodes;j++)

Path[i][j] = Adj[i][j];

**//Нахождение следующих значений матрицы Path.**

**for** (**int** k=0;k<MaxNodes;k++)

**for** (i=0;i<MaxNodes;i++)

**if** (Path[i][k]==1)

**for** (**int** j=0;j<MaxNodes;j++)

Path[i][j] = (Path[i][j] || Path[k][j]);

}

**void** Warshall::Vyvod()

**//Вывод матрицы достижимости.**

{

cout << "Матрица достижимости:\n";

**for** (**int** i=0;i<MaxNodes;i++)

{

**for** (**int** j=0;j<MaxNodes;j++)

cout << Path[i][j] << " ";

cout << endl;

}

}

**void** main()

{

Warshall A;

A.Vvod();

A.TransClose();

A.Vyvod();

}

Текст этой программы можно взять [здесь](http://khpi-iip.mipk.kharkiv.edu/library/datastr/book_sod/kgsu/din100_1.zip).

Заметим, что описанный алгоритм обрабатывает матрицу **Adj** по столбцам. ***Уоррен*** [3, с.320] предложил изящный двухпроходной ***строкоориентированный алгоритм***. В этом алгоритме при обработке вершины, например **i**, в первом проходе обрабатываются только ребра, связанные с вершинами, меньшими **i**, а во втором проходе - только ребра, связанные с вершинами, большими **i**. Другими словами, алгоритм преобразует матрицу смежности **Adj** графа **G** в матрицу достижимости, обрабатывая в первом проходе только элементы матрицы, расположенные ниже ее главной диагонали, а во втором проходе - только элементы матрицы, расположенные выше ее главной диагонали. Таким образом, при каждом проходе обрабатывается не более **n\*(n-1)/2** ребер.

Программа 2. ***Приведем реализацию алгоритма Уоррена на языке* C++**.

**#include** <iostream.h>

**#define** MaxNodes 4

**class** Warren

{

**private**:

**unsigned** Adj[MaxNodes][MaxNodes]; **//Матрица смежностей.**

**unsigned** Path[MaxNodes][MaxNodes]; **//Матрица достижимости.**

**public**:

**void** Vvod();

**void** TransClose();

**void** Vyvod();

};

**void** Warren::Vvod()

{

**//Ввод матрицы смежностей заданного графа.**

cout <<"Вводите элементы матрицы смежностей по строкам:\n";

**for** (**int** i=0;i<MaxNodes;i++)

**for** (**int** j=0;j<MaxNodes;j++)

{

cout <<"Введите Adj["<< (i+1) << "]["<< (j+1) << "]: ";

cin >> Adj[i][j];

}

}

**void** Warren::TransClose()

**//Вычисление матрицы достижимости.**

{

**//Инициализация матрицы Path матрицей смежностей Adj.**

**for** (**int** i=0;i<MaxNodes;i++)

**for** (**int** j=0;j<MaxNodes;j++)

Path[i][j] = Adj[i][j];

**//Нахождение следующих значений матрицы Path.**

**for** (i=1;i<MaxNodes;i++)

**for** (**int** j=0;j<i;j++)

**if** (Path[i][j]==1)

**for** (**int** k=0;k<MaxNodes;k++)

Path[i][k] = (Path[i][k] || Path[j][k]);

**for** (i=0;i<MaxNodes-1;i++)

**for** (**int** j=i+1;j<MaxNodes;j++)

**if** (Path[i][j]==1)

**for** (**int** k=0;k<MaxNodes;k++)

Path[i][k] = (Path[i][k] || Path[j][k]);

}

**void** Warren::Vyvod()

**//Вывод матрицы достижимости.**

{

cout << "Матрица достижимости:\n";

**for** (**int** i=0;i<MaxNodes;i++)

{

**for** (**int** j=0;j<MaxNodes;j++)

cout << Path[i][j] << " ";

cout << endl;

}

}

**void** main()

{

Warren A;

A.Vvod();

A.TransClose();

A.Vyvod();

}

Текст этой программы можно взять [здесь](http://khpi-iip.mipk.kharkiv.edu/library/datastr/book_sod/kgsu/din100_2.zip).

(1) Трамбле Ж., Соренсон П. Введение в структуры данных. - М.: Машиностроение, 1982. - 784 с.

(2) Tenenbaum A., Augenstein M. Data Structures Using Pascal. Englewood Cliffs. - N.Y.: Prentice-Hall, Inc. 1981.

(3) Свами М., Тхуласираман К. Графы, сети и алгоритмы. - М.: Мир, 1984. - 454 с.

На следующем шаге мы рассмотрим ***применение алгоритма Уоршалла***.

## Применение алгоритма Уоршалла. Вычисление длин кратчайших путей между вершинами

На этом шаге мы рассмотрим ***алгоритм нахождения кратчайших путей между вершинами графа***.

Алгоритм Уоршалла может быть модифицирован с целью получения матрицы, содержащей ***длины кратчайших путей между вершинами*** [1, с.441].

Идея этого алгоритма следующая [2,с.131]. Обозначим через **d(m)ij** длину кратчайшего из путей из **vi** в **vj** с промежуточными вершинами в множестве **{v1, ..., vm}**.

Тогда имеем следующие уравнения, объединенные в систему:

d(0)ij = aij

d(m)ij = min(d(m)ij, d(m)im + d(m)mj)

Обоснование второго уравнения достаточно простое. Рассмотрим кратчайший путь из **vi** в **vj** c промежуточными вершинами из множества **{v1, ..., vm,vm+1}**. Если этот путь не содержит **vm+1**, то деля путь на отрезки от **vi** до **vm+1** и от **vm+1** до **vj**, получаем равенство

d(m+1)ij = d(m)im + d(m)mj).

Вышеприведенные уравнения дают возможность легко вычислить расстояния

d(vi,vj) = d(n)ij, 1 <= i, j <= n.

Для машинной реализации алгоритма предположим, что **A** - матрица смежности графа. Заменим все нулевые элементы **A** на "бесконечность" или на некоторое "очень большое число", которое в данной матрице обозначим через **B**. Требуемая матрица длин минимальных путей формируется следующим алгоритмом, записанным на языке C++.

Программа. ***Вычисление матрицы минимальных длин путей***.

**#include** <iostream.h>

**#define** MaxNodes 4

**#define** B 1000

**class** Warshall

{

**private**:

**unsigned** Adj[MaxNodes][MaxNodes]; **//Матрица смежностей.**

**unsigned** C[MaxNodes][MaxNodes]; **//Матрица достижимости.**

**public**:

**void** Vvod();

**void** MinDlin();

**void** Vyvod();

};

**void** Warshall::Vvod()

**//Ввод матрицы смежностей заданного графа и ее "исправление".**

{

**//Ввод матрицы смежностей заданного графа.**

cout <<"Вводите элементы матрицы смежностей по строкам:\n";

**for** (**int** i=0;i<MaxNodes;i++)

**for** (**int** j=0;j<MaxNodes;j++)

{

cout <<"Введите Adj["<< (i+1) << "]["<< (j+1) << "]: ";

cin >> Adj[i][j];

**if** (Adj[i][j]==0) C[i][j] = B;

**else** C[i][j] = Adj[i][j];

}

}

**void** Warshall::MinDlin()

**//Вычисление матрицы минимальных длин путей.**

{

**for** (**int** k=0;k<MaxNodes;k++)

**for** (**int** i=0;i<MaxNodes;i++)

**for** (**int** j=0;j<MaxNodes;j++)

**if** ( C[i][j]>C[i][k]+C[k][j] )

C[i][j] = C[i][k]+C[k][j];

}

**void** Warshall::Vyvod()

**//Вывод матрицы минимальных длин путей.**

{

cout << "Матрица минимальных длин путей:\n";

**for** (**int** i=0;i<MaxNodes;i++)

{

**for** (**int** j=0;j<MaxNodes;j++)

cout << C[i][j] << " ";

cout << endl;

}

}

**void** main()

{

Warshall A;

A.Vvod();

A.MinDlin();

A.Vyvod();

}

Текст этой программы можно взять [здесь](http://khpi-iip.mipk.kharkiv.edu/library/datastr/book_sod/kgsu/din101_1.zip).

Очевидно, что сложность алгоритма есть **O(n3)**. В монографии [2, с.132] отмечено, что "для общего случая (т.е. без предположения о неотрицательности весов или о бесконтурности графа) ***не известен*** ни один алгоритм нахождения расстояния между одной фиксированной парой вершин, который был бы значительно эффективнее алгоритма нахождения расстояний между всеми парами вершин."

***Замечание.*** *Авторами приведенного алгоритма являются* ***Уоршалл*** *и* ***Флойд*** *(****Floyd R.W.****).*

(1) Трамбле Ж., Соренсон П. Введение в структуры данных. - М.: Машиностроение, 1982. - 784 с.

(2) Липский В. Комбинаторика для программистов: Пер. с польск. - М.: Мир, 1988. - 213 с.

На следующем шаге мы рассмотрим ***алгоритм отыскания компонент сильной связности***.

## Применение алгоритма Уоршалла. Отыскание компонент сильной связности

На этом шаге мы рассмотрим ***еще одну область применимости алгоритма Уоршалла***.

Рассмотрим матричный алгоритм отыскания ***компонент сильной связности (бикомпонент)*** ориентированного графа, следуя монографии [1, с.54-55].

***Определение.***

Ориентированный граф называется ***сильно связным***, если для любой пары вершин каждая из них достижима из другой [1, с.11].

Сильно связный ***подграф*** ориентированного графа называется ***зоной***.

Зона, максимальная относительно включения вершин, называется ***компонентой сильной связности (бикомпонентой)***.

Знание матрицы достижимости позволяет выявить все компоненты сильной связности в ориентированном графе.

Определим ***поэлементное (адамарово) произведение*** матриц **B=(bij)** и **C=(cij)** по правилу: **B \*C=(bij \* cij)**.

Тогда вершины бикомпоненты ориентированного графа, содержащей вершину **xi**, определяются единичными элементами **i**-й строки матрицы **R \* RT**, где **RT** - транспонированная матрица достижимости.

Отсюда следует, что, вычислив поэлементное произведение матриц **R** и **RT** и разбив все ненулевые строки этого произведения на группы одинаковых строк, мы можем определить множество вершин каждой бикомпоненты, поскольку номера строк ***однозначно*** определяют номера вершин, входящих в бикомпоненту.

Пример. ***Нахождение бикомпонент ориентированного графа.***

**#include** <iostream.h>

**#define** MaxNodes 4

**class** Warshall

{

**private**:

**unsigned** Adj[MaxNodes][MaxNodes]; **//Матрица смежностей.**

**unsigned** Path[MaxNodes][MaxNodes]; **//Матрица достижимости.**

**unsigned** PathT[MaxNodes][MaxNodes];**//Транспонированная матрица достижимости.**

**unsigned** Adam[MaxNodes][MaxNodes]; **//Результат адамарова**

**//произведения Path\*PathT.**

**public**:

**void** Vvod();

**void** TransClose();

**void** Vyvod\_D();

**void** Trans();

**void** Adamar();

**void** Vyvod\_A();

};

**void** Warshall::Vvod()

{

**//Ввод матрицы смежностей заданного графа.**

cout <<"Вводите элементы матрицы смежностей по строкам:\n";

**for** (**int** i=0;i<MaxNodes;i++)

**for** (**int** j=0;j<MaxNodes;j++)

{

cout <<"Введите Adj["<< (i+1) << "]["<< (j+1) << "]: ";

cin >> Adj[i][j];

}

}

**void** Warshall::TransClose()

**//Вычисление матрицы достижимости.**

{

**//Инициализация матрицы Path матрицей смежностей Adj.**

**for** (**int** i=0;i<MaxNodes;i++)

**for** (**int** j=0;j<MaxNodes;j++)

Path[i][j] = Adj[i][j];

**//Нахождение следующих значений матрицы Path.**

**for** (**int** k=0;k<MaxNodes;k++)

**for** (i=0;i<MaxNodes;i++)

**if** (Path[i][k]==1)

**for** (**int** j=0;j<MaxNodes;j++)

Path[i][j] = (Path[i][j] || Path[k][j]);

}

**void** Warshall::Vyvod\_D()

**//Вывод матрицы достижимости.**

{

cout << "Матрица достижимости:\n";

**for** (**int** i=0;i<MaxNodes;i++)

{

**for** (**int** j=0;j<MaxNodes;j++)

cout << Path[i][j] << " ";

cout << endl;

}

}

**void** Warshall::Trans()

**//Транспонирование матрицы Path и помещение результата в**

**//матрицу PathT.**

{

**unsigned** k,r;

**for** (**int** i=0;i<MaxNodes;i++)

**for** (**int** j=0;j<MaxNodes;j++)

{

k = Path[i][j]; r = Path[j][i];

PathT[j][i] = k; PathT[i][j] = r;

}

}

**void** Warshall::Adamar()

**//Нахождение адамарова произведения PathхPathT.**

{

**for** (**int** i=0;i<MaxNodes;i++)

**for** (**int** j=0;j<MaxNodes;j++)

Adam[i][j] = Path[i][j]\*PathT[i][j];

}

**void** Warshall::Vyvod\_A()

**//Вывод адамарова произведения.**

{

cout << "Адамарово произведение:\n";

**for** (**int** i=0;i<MaxNodes;i++)

{

**for** (**int** j=0;j<MaxNodes;j++)

cout << Adam[i][j] << " ";

cout << endl;

}

}

**void** main()

{

Warshall A;

A.Vvod();

A.TransClose();

A.Vyvod\_D();

A.Trans();

A.Adamar();

A.Vyvod\_A();

}

Текст этой программы можно взять [здесь](http://khpi-iip.mipk.kharkiv.edu/library/datastr/book_sod/kgsu/din102_1.zip).

***Замечания.***

1. *Приведенный простейший матричный алгоритм отыскания бикомпонент принадлежит, по-видимому,* ***С.Рамамурти*** *[1, с.119].*
2. *[1, с.55]. Более эффективные алгоритмы отыскания бикомпонент основаны на обходе графа в глубину.*

(1) Евстигнеев В.А. Применение теории графов в программировании. -М.: Hаука, 1985. - 352 с.

На следующем шаге мы рассмотрим вопросы, связанные с ***определением рекурсивности процедуры***.

## Применение алгоритма Уоршалла. Определение рекурсивности подпрограммы

На этом шаге мы рассмотрим ***способ определения рекурсивности подпрограммы***.

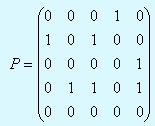
Сейчас мы покажем, как путевая матрица ориентированного графа может быть использована для решения вопроса о том, являются ли определенные процедуры в программе ***рекурсивными***.

В некоторых языках программирования программист может задать рекурсивность процедуры в явной форме. В других языках для того, чтобы определить, какие из процедур являются рекурсивными, можно использовать идеи, вытекающие из теории графов.

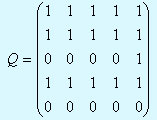
Вначале необходимо заметить, что ***рекурсивная процедура необязательно вызывает себя непосредственно***. Если процедура **P1** вызывает **P2**, процедура **P2** вызывает **P3**, ..., процедура **Pn-1** вызывает **Pn** и процедура **Pn** вызывает **P1**, то процедура **P1** является ***рекурсивной***. Подобная форма рекурсии называется ***косвенной***.

Пусть **P={P1, P2, P3, ..., Pn}** - набор процедур в программе. Построим ориентированный граф, состоящий из вершин, соответствующих элементам **P**. Ребро из **Pi** в **Pj** проведем в графе в том случае, когда процедура **Pi** вызывает процедуру **Pj**.

Например, пусть матрица смежностей, соответствующая вызовам, проводимым некоторым набором процедур, имеет вид:



Процедура **Pi** является рекурсивной, если в графе имеется цикл, содержащий **Pi**. Такие циклы могут быть обнаружены путем анализа ***диагональных элементов матрицы достижимости* Q *графа***. Так, **Pi** является рекурсивной процедурой, если **Qii=1**. Матрица **Q** может быть построена с помощью [программы 1 из шага 100](http://khpi-iip.mipk.kharkiv.edu/library/datastr/book_sod/kgsu/din_0100.html#1). В приведенном выше примере она имеет вид:



откуда следует, что процедуры **P1, P2** и **P4** являются ***рекурсивными***.

***Замечание [1, с.132-133]****.*

*С задачей определения кратчайших путей в графе тесно связана* ***задача транзитивного замыкания бинарного отношения****.*

*Вспомним, что под* ***бинарным отношением на множестве V*** *мы понимаем множество* ***E****, являющееся подмножеством* ***V\*V****.*

*Отношение является* ***транзитивным****, если удовлетворяется условие: если* ***(x,y)*** *принадлежит* ***E*** *и* ***(y,z)*** *принадлежит* ***E****, то* ***(x,z)*** *принадлежит* ***E*** *для произвольных* ***x,y,z****, принадлежащих* ***E****.*

*Для произвольного такого отношения мы определяем*

*E\* = {(x,y): в (V,E) существует путь ненулевой длины из x в y}.*

*Нетрудно заметить, что* ***E\**** *- транзитивное отношение на множестве* ***V*** *и* ***E*** *принадлежит* ***E\*****. Более того,* ***E\*****является наименьшим транзитивным отношением, содержащим* ***E****, т.е. для произвольного транзитивного отношения* ***F****, такого, что* ***E*** *принадлежит* ***F*** *выполняется включение* ***E\**** *принадлежит* ***F****.*

*Отношение* ***E\**** *называется* ***транзитивным замыканием отношения E****.*

*Отметим, что бинарное отношение* ***E****, принадлежащее* ***V\*V****, можно однозначно представить* ***ориентированным графом G=(V,E)****.*

*Ориентированный граф называется* ***транзитивным****, если из существования дуг* ***(xi,xj)*** *и* ***(xj,xk)*** *следует существование дуги* ***(xi,xk)****.*

*Транзитивным замыканием графа* ***G=(X,A)*** *является граф* ***Gtc=(X,AUA')****, где* ***A'*** *является минимально возможным множеством дуг, необходимых для того, чтобы граф* ***Gtc*** *был транзитивным. Так как путь от* ***xi*** *к* ***xj*** *в графе* ***G****должен соответствовать дуге* ***(xi,xj)*** *в* ***Gtc****, то совершенно очевидно, что матрица достижимости* ***R*** *графа* ***G*** *почти полностью совпадает с матрицей смежностей* ***A*** *графа* ***Gtc*** *- надо только в матрице* ***A*** *поставить на главной диагонали единицы.*

(1) Липский В. Комбинаторика для программистов: Пер. с польск. - М.: Мир, 1988. - 213 с.

На следующем шаге мы рассмотрим ***контуры в ориентированных графах***.

## Кратчайшие пути между всеми парами вершин. Контуры в ориентированных графах

На этом шаге мы рассмотрим ***отыскание контуров в ориентированных графах***.

Одной из важнейших задач, связанных с контурами является задача ***нахождения множества всех контуров*** [1, с.105]. Трудность ее состоит прежде всего в том, что число контуров ориентированного графа может быть экспоненциально большим относительно числа вершин. Поэтому при разработке алгоритмов внимание обращается не на полную трудоемкость алгоритма, а на относительную, т.е. на трудоемкость, приходящуюся на один контур.

***Алгоритм отыскания множества вершин, принадлежащих контуру заданной длины***

Алгоритм использует матрицу смежности **A(G)** и матрицу **Ak**, если длина контура равна **k**. Выберем некоторое **i**, такое, что **aii(k)=1**. Это означает, что вершина **vi** принадлежит контуру длины **k**.

Тогда вершина **vj** принадлежит тому же контуру, если выполняются следующие три условия:

* **ajj(k)=1**;
* для любого **n** **aij(n)=1**, т.е. существует путь длины **n** из **vi** в **vj**;
* **aji(k-n)=1**, т.е. существует путь длины **k-n** из **vj** в **vi**.

Таким образом, для каждой вершины **i** графа мы легко можем построить множество вершин, каждый элемент которого принадлежит некоторому контуру длины **k**.

Пример.

**#include** <iostream.h>

**#define** MaxNodes 4

**#define** Stepen 10

**class** Warshall

{

**private**:

**unsigned** Adj[MaxNodes][MaxNodes]; **//Матрица смежностей.**

**//Массив степеней матрицы смежностей.**

**unsigned** AdjN[Stepen][MaxNodes][MaxNodes];

**void** Power(**int**);

**public**:

**void** Vvod();

**void** Step();

**void** Kontur();

};

**void** Warshall::Vvod()

{

**//Ввод матрицы смежностей заданного графа.**

cout <<"Вводите элементы матрицы смежностей по строкам:\n";

**for** (**int** i=0;i<MaxNodes;i++)

**for** (**int** j=0;j<MaxNodes;j++)

{

cout <<"Введите Adj["<< (i+1) << "]["<< (j+1) << "]: ";

cin >> Adj[i][j];

}

}

**void** Warshall::Step()

**//Вычисление степеней матрицы смежностей.**

{

**for** (**int** l=0;l<Stepen;l++)

{

Power(l);

}

}

**void** Warshall::Power(**int** n)

**//Возвращает значение n-й степени матрицы A.**

{

**unsigned** Z[MaxNodes][MaxNodes];

**unsigned** C[MaxNodes][MaxNodes];

**unsigned** Val;

**for** (**int** i=0;i<MaxNodes;i++)

**for** (**int** j=0;j<MaxNodes;j++) C[i][j]=Adj[i][j];

**for** (**int** m=0;m<n-1;m++)

{

**for** (**int** i=0;i<MaxNodes;i++)

**for** (**int** j=0;j<MaxNodes;j++)

{

Val = 0;

**for** (**int** k=0;k<MaxNodes;k++)

Val = Val || (Adj[i][k] && C[k][j]);

Z[i][j] = Val;

}

**for** (i=0;i<MaxNodes;i++)

**for** (**int** j=0;j<MaxNodes;j++) C[i][j]=Z[i][j];

}

**for** (i=0;i<MaxNodes;i++)

**for** (**int** j=0;j<MaxNodes;j++)

AdjN [n][i][j] = C[i][j];

}

**void** Warshall::Kontur()

**//Отыскание контуров заданной длины.**

{

**unsigned** n;

cout << "Вводите длину контура: ";

cin >> n;

**for** (**int** m=1;m<n;m++)

{

**for** (**int** i=0;i<MaxNodes;i++)

**if** ( AdjN [m][i][i]==1 )

**//Вершина i+1 принадлежит контуру длины n.**

{

cout << "Вершина " << (i+1) <<

" образует контуры длины " << (m+1) << " с вершинами

из множества: {";

**for** (**int** j=0;j<MaxNodes;j++)

{

**if** ( AdjN[m][j][j]==1 )

**//Вершина j принадлежит контуру длины m.**

**for** (**int** l=0;l<m;l++)

**if** ( AdjN[l][i][j]==1 && m-l>0

&& AdjN[m-l][j][i]==1 )

{

cout << (j+1) << ' ';

goto Metka;

}

Metka:;

}

cout << '}' << endl;

}

cout << endl;

}

}

**void** main()

{

Warshall A;

A.Vvod();

A.Step();

A.Kontur();

}

Текст этой программы можно взять [здесь](http://khpi-iip.mipk.kharkiv.edu/library/datastr/book_sod/kgsu/din104_1.zip).

(1) Евстигнеев В.А. Применение теории графов в программировании. - М.: Hаука, 1985. - 352 с.

На следующем шаге мы начнем рассматривать вопросы ***связности графа***.

## Связность. Вычисление компонент связности

На этом шаге мы изменим алгоритм обхода графа в ширину для ***вычисления конпонент связности***.

С помощью изложенных в шагах [96](http://khpi-iip.mipk.kharkiv.edu/library/datastr/book_sod/kgsu/din_0096.html) и [97](http://khpi-iip.mipk.kharkiv.edu/library/datastr/book_sod/kgsu/din_0097.html) алгоритмов обхода графа можно ответить на некоторые вопросы относительно структуры графа. Мы рассмотрим здесь две задачи, связанные с обходом графа:

* ***вычисление компонент связности графа*** и
* ***построение остовного дерева связного графа***.

***Определение.***

Рассмотрим некоторое семейство подграфов графа **G**.

Граф **Hmax** из этого семейства называется ***максимальным***, если он не содержится ни в каком другом графе из рассматриваемого семейства.

Максимальный связный подграф графа **G** называется ***связной компонентой* G** ***(компонентой связности* G)**. В связном графе имеется единственная связная компонента, совпадающая с самим графом.

***Вычисление компонент связности***

Отметим, что алгоритм [***обхода графа в глубину***](http://khpi-iip.mipk.kharkiv.edu/library/datastr/book_sod/kgsu/din_0096.html) можно легко модифицировать так, чтобы он вычислял ***связные компоненты*** этого графа.

Пример 1. Иллюстрация использования обхода графа в глубину для нахождения ***всех его связных компонент. Граф представлен структурой Вирта***.

**#include** <iostream.h>

**#define** TRUE 1

**#define** FALSE 0

**typedef** **int** Boolean;

**typedef** **struct** L \*Lref; **// Тип: указатель на заголовочный узел.**

**typedef** **struct** T \*Tref; **// Тип: указатель на дуговой узел.**

**//Описание типа заголовочного узла.**

**typedef** **struct** L

{

**int** Key; **//Имя заголовочного узла.**

**int** Count; **//Количество предшественников.**

Boolean Flag; **//Флаг посещения узла при обходе.**

Tref Trail; **//Указатель на список смежности.**

Lref Next; **//Указатель на следующий узел в списке заголовочных узлов.**

} Leader;

**//Описание типа дугового узла.**

**typedef** **struct** T

{

Lref Id;

Tref Next;

} Trailer;

**//Описание типа узла стека.**

**typedef** Tref TipElement;

**typedef** **struct** Zveno \*svqz;

**typedef** **struct** Zveno

{

TipElement Element; **//Указатель на список смежности.**

svqz Sled;

} St;

**class** Spisok

{

**private**:

Lref Head; **//Указатель на голову списка заголовочных узлов.**

Lref Tail; **//Указатель на фиктивный элемент**

**// в конце списка заголовочных узлов.**

**void** SearchGraph (int, Lref \*);

**public**:

Spisok() {**//Инициализация списка заголовочных узлов.**

Head = Tail = **new** (Leader); }

Lref GetHead() { **return** Head; }

Lref GetTail() { **return** Tail; }

**void** MakeGraph ();

**void** PrintGraph ();

**void** Depth\_First\_Search (Lref);

**void** LinkComp();

};

**void** main ()

{

Spisok A;

Lref t; **//Рабочий указатель для перемещения**

**// по списку заголовочных звеньев.**

**//Построение графа и вывод его структуры Вирта.**

A.MakeGraph ();

A.PrintGraph (); cout<<endl;

**//Рекурсивный обход графа в глубину.**

cout<<"Результат рекурсивного обхода...\n";

t = A.GetHead();

**while** (t!=A.GetTail())

{ (\*t).Flag = TRUE; t = (\*t).Next; }

A.Depth\_First\_Search (A.GetHead()); cout<<endl;

**//Вывод компонент связности.**

cout<<"Компоненты связности:\n";

A.LinkComp();

}

**void** Spisok::SearchGraph (**int** w, Lref \*h)

**//Функция возвращает указатель на заголовочный узел**

**//с ключом w в графе, заданном структурой Вирта с указателем Head.**

{

\*h = Head; (\*Tail).Key = w;

**while** ((\*\*h).Key!=w) \*h = (\*\*h).Next;

**if** (\*h==Tail)

**//В списке заголовочных узлов нет узла с ключом w.**

**//Поместим его в конец списка Head.**

{ Tail = **new** (Leader); (\*\*h).Count = 0;

(\*\*h).Trail = **NULL**; (\*\*h).Next = Tail; }

}

**void** Spisok::MakeGraph ()

**//Функция возвращает указатель Head на структуру**

**//Вирта, соответствующую ориентированному графу.**

{

**int** x,y;

Lref p,q; **//Рабочие указатели.**

Tref t,r; **//Рабочие указатели.**

Boolean Res; **//Флаг наличия дуги.**

cout<<"Вводите начальную вершину дуги: ";

cin>>x;

**while** (x!=0)

{

cout<<"Вводите конечную вершину дуги: "; cin>>y;

**//Определим, существует ли в графе дуга (x,y)?**

SearchGraph (x, &p); SearchGraph (y,&q);

r = (\*p).Trail; Res = FALSE;

**while** ((r!=**NULL**)&&(!Res))

**if** ((\*r).Id==q) Res = TRUE;

**else** r = (\*r).Next;

**if** (!Res) **//Если дуга отсутствует, то поместим её в граф.**

{ t = **new** (Trailer); (\*t).Id = q;

(\*t).Next = (\*p).Trail; (\*p).Trail = t; (\*q).Count++; }

cout<<"Вводите начальную вершину дуги: "; cin>>x;

}

}

**void** Spisok::PrintGraph ()

**//Вывод структуры Вирта, заданной указателем**

**//Head и соответствующей ориентированному графу.**

{

Lref p; **//Рабочий указатель.**

Tref q; **//Рабочий указатель.**

p = Head;

**while** (p!=Tail)

{

cout<<"("<<(\*p).Key; q = (\*p).Trail;

**while** (q!=**NULL**)

{ cout<<(\*(\*q).Id).Key; q = (\*q).Next; }

cout<<")"; p = (\*p).Next; cout<<" ";

}

}

**void** Spisok::Depth\_First\_Search (Lref r)

**//Рекурсивный обход графа в глубину. r - указатель**

**//на структуру Вирта.**

{

Tref t;

t = (\*r).Trail; cout<<(\*r).Key; (\*r).Flag = FALSE;

**while** (t!=**NULL**)

{ **if** ((\*(\*t).Id).Flag) Depth\_First\_Search ((\*t).Id); t = (\*t).Next; }

}

**void** Spisok::LinkComp()

**//Определение компонент связности в графе, заданном**

**//структурой Вирта с указателем Head.**

{

Lref t = Head;

**while** (t !=Tail)

{ t->Flag = TRUE; t = t->Next; }

t = Head;

**while** ( t!=Tail )

{

**if** ( t->Flag )

{ Depth\_First\_Search (t); cout << endl; }

t = t->Next;

}

}

Текст этой программы можно взять [здесь](http://khpi-iip.mipk.kharkiv.edu/library/datastr/book_sod/kgsu/din105_1.zip).

Результат работы приложения изображен на рисунке 1:

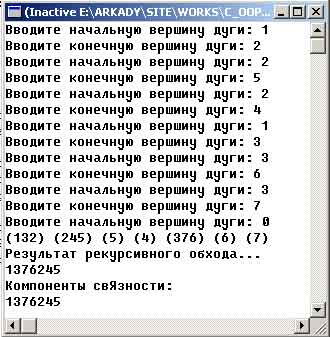


Рис.1. Результат работы приложения

***Для вычисления компонент связности графа на языке* LISP** воспользуемся уже написанной ранее [***функцией*DEPTHFIRST**](http://khpi-iip.mipk.kharkiv.edu/library/datastr/book_sod/kgsu/din_0096.html#1) [1].

Рассмотрим еще раз работу функции **DEFI**. Заметим, что когда к списку просмотренных вершин **VISITED**добавляется новая вершина, список **PATH** представляет собой путь из этой вершины в начальную. Поэтому по окончанию работы в списке **VISITED** содержатся только те вершины, которые можно соединить с начальной. Можно показать [2], что в этом списке содержатся все вершины, обладающие этим свойством.

Таким образом, можно сказать, что функция **DEPTHFIRST** вычисляет список вершин связной компоненты вершины **ROOT**, и теперь мы готовы решить задачу о построении всех компонент связности данного графа.

Проще всего это сделать в два этапа.

1. Сначала ***построим списки вершин компонент связности***.

***Программа [3, с.129-130].***

(DEFUN CONNLISTS (LAMBDA (GRAPH)

**; Построение списка списков вершин компонент связности**

**; GRAPH - граф, представленный структурой смежности в**

**; виде ассоциативного списка**

(CONNLSTS GRAPH NIL)

))

**; -----------------------------------**

(DEFUN CONNLSTS (LAMBDA (GRAPH LISTS)

(COND ( (**NULL** GRAPH) LISTS )

( T (COND ( (**NULL** (LMEMBER (CAAR GRAPH) LISTS))

(CONNLSTS

(CDR GRAPH)

(CONS (DEPTHFIRST GRAPH (CAAR GRAPH))

LISTS)) )

( T (CONNLSTS (CDR GRAPH) LISTS) )) )

)

))

**; -----------------------------------**

(DEFUN LMEMBER (LAMBDA (VERTEX LISTS)

**; Функция проверяет, содержится ли вершина VERTEX в**

**; каком-либо из списков, составляющих LISTS**

(AND LISTS

(OR (MEMBER VERTEX (CAR LISTS))

(LMEMBER VERTEX (CDR LISTS))))

))

**; ------------------------------------**

(DEFUN DEPTHFIRST (LAMBDA (GRAPH ROOT)

**; Обход графа в глубину:**

**; GRAPH - граф, представленный структурой смежности в**

**; виде ассоциативного списка,**

**; ROOT - вершина, с которой начинается обход графа,**

**; Результат: список вершин графа в порядке посещения в глубину**

(COND ( (**NULL** GRAPH) NIL )

( T (DEFI GRAPH (LIST ROOT) (LIST ROOT)) ))

))

**; --------------------------------------**

(DEFUN DEFI (LAMBDA (GRAPH VISITED PATH)

**; GRAPH - граф, представленный структурой смежности в**

**; виде ассоциативного списка,**

**; VISITED - список уже посещенных вершин,**

**; PATH - список вершин, определяющий путь посещения**

(COND ( (**NULL** PATH) (REVERSE VISITED) )

( T (COND ( (**NULL** (EXPND GRAPH VISITED (CAR PATH)))

(DEFI GRAPH VISITED (CDR PATH)) )

( T (DEFI GRAPH

(CONS (EXPND GRAPH VISITED (CAR PATH))

VISITED)

(CONS (EXPND GRAPH VISITED (CAR PATH))

PATH)) )) )

)

))

**; -----------------------------------------**

(DEFUN EXPND (LAMBDA (GRAPH VISITED VERTEX)

**; Выбор в графе GRAPH следующей еще не просмотренной**

**; вершины, смежной с вершиной VERTEX**

(COND ( (**NULL** (NEIGHBOUR3 VERTEX GRAPH)) NIL )

( T (FIRSTNOTVISITED VISITED (NEIGHBOUR3 VERTEX GRAPH)) )

)

))

**; --------------------------------------------**

(DEFUN FIRSTNOTVISITED (LAMBDA (VISITED VLIST)

**; Поиск первой непосещенной вершины в списке VLIST**

**; VISITED - список уже посещенных вершин**

(COND ( (**NULL** VLIST) NIL )

( T (COND ( (**NULL** (MEMBER (CAR VLIST) VISITED))

(CAR VLIST) )

( T (FIRSTNOTVISITED VISITED (CDR VLIST)) )) )

)

))

**; ---------------------------------**

(DEFUN NEIGHBOUR3 (LAMBDA (X GRAPH)

**; Функция возвращает список вершин графа GRAPH, смежных с**

**; вершиной X**

(COND ( (**NULL** (ASSOC X GRAPH)) NIL )

( T (CDR (ASSOC X GRAPH)) ))

))

Текст этой библиотеки можно взять [здесь](http://khpi-iip.mipk.kharkiv.edu/library/datastr/book_sod/kgsu/din105_2.zip).

Тестовые примеры:

$ (CONNLISTS '((1 . (2 3)) (2 . (3)) (5 . ())))

((5) (1 2 3))

$ (CONNLISTS '((1 . (2 3)) (2 . (3)) (5 . ()) (6 . ())))

((6) (5) (1 2 3))

$ (CONNLISTS '((1 . (2 3)) (2 . (3)) (5 . (6))))

((5 6) (1 2 3))

$ (CONNLISTS '((1 . (2 3 5)) (2 . (3)) (5 . (6))))

((1 2 3 5 6))

2. Теперь не составляет труда написать функцию, возвращающую структуры смежности, соответствующие компонентам связности. Для этого мы воспользуемся следующими вспомогательными функциями:

(DEFUN F2 (LAMBDA (GRAPH LST)

**; Построение структур смежности всех компонент связности**

**; GRAPH - граф, представленный структурой смежности в**

**; виде ассоциативного списка,**

**; LST - список списков компонент связности**

(COND ( (**NULL** LST) NIL )

( T (CONS (F1 GRAPH (CAR LST))

(F2 GRAPH (CDR LST))) ))

))

**; ---------------------------**

(DEFUN F1 (LAMBDA (GRAPH LST)

(COND ( (**NULL** GRAPH) NIL )

( (MEMBER (CAAR GRAPH) LST)

(CONS (CAR GRAPH) (F1 (CDR GRAPH) LST)) )

( T (F1 (CDR GRAPH) LST) ))

))

Текст этой библиотеки можно взять [здесь](http://khpi-iip.mipk.kharkiv.edu/library/datastr/book_sod/kgsu/din105_3.zip).

Тестовый пример:

$ (F2 '((1 . (2 3)) (4 . (5 6)) (5 . (4)) (3 . (2 1)))

'((1 2 3) (4 5 6)))

(((1 2 3) (3 2 1)) ((4 5 6) (5 4)))

(1) Крюков А.П., Радионов А.Я., Таранов А.Ю., Шаблыгин Е.М. Программирование на языке R-Лисп. - М.: Радио и связь, 1991. - 192 с.

(2) Баррон Д. Рекурсивные методы в программировании. - М.: Мир, 1974. - 80 с.

(3) Turbo Pascal Version 3.0 Reference Manual. Borland International, 1985.

На следующем шаге мы рассмотрим ***нахождение компонент двусвязности***.

## Связность. Нахождение компонент двусвязности

На этом шаге мы рассмотрим ***алгоритм нахождения компонент двусвязности***.

***Определение [1, с.101].***

Вершину **a** неориентированного графа **G=(V,E)** будем называть ***точкой сочленения***, если удаление этой вершины и всех инцидентных ей ребер ведет к увеличению числа компонент связности графа.

Неориентированный граф называется ***двусвязным***, если он связный и не содержит точек сочленения.

Произвольный максимальный двусвязный подграф графа **G** называется ***компонентой двусвязности*** или***блоком*** этого графа.

Двусвязность графа - очень желательный признак для некоторых приложений. Представим себе, что вершины графа изображают узлы некоторой информационной сети, а ребра соответствуют линиям передачи. Если наш граф двусвязный, то выход из строя отдельного узла **w** никогда не приведет к потере соединения между любыми двумя узлами, отличными от **w**. Знание блоков графа также очень важно, если принять во внимание то, что многие графовые задачи, такие как нахождение всех элементарных циклов или установление факта планарности графа (граф называется планарным, если его можно так начертить на плоскости, чтобы никакие два ребра не пересекались), приводят естественным путем к аналогичным задачам для блоков данного графа.

Нахождение точек сочленения и блоков графа является классической задачей, которую можно эффективно решить ***"погружением"*** в процедуру обхода графа в глубину.

Пример.

**#include** <iostream.h>

**#define** TRUE 1

**#define** FALSE 0

**typedef** **int** Boolean;

**typedef** **struct** L \*Lref; **// Тип: указатель на заголовочный узел.**

**typedef** **struct** T \*Tref; **// Тип: указатель на дуговой узел.**

**//Описание типа заголовочного узла.**

**typedef** **struct** L

{

**int** Key; **//Имя заголовочного узла.**

**int** Count; **//Количество предшественников.**

**int** WGN; **//Характеристика узла.**

Tref Trail; **//Указатель на список смежности.**

Lref Next; **//Указатель на следующий узел в списке заголовочных узлов.**

} Leader;

**//Описание типа дугового узла.**

**typedef** **struct** T

{

Lref Id;

Tref Next;

} Trailer;

**//Описание типа узла стека.**

**typedef** Lref TipElement;

**typedef** **struct** Zveno \*svqz;

**typedef** **struct** Zveno

{

TipElement Element1;

TipElement Element2;

svqz Sled;

} St;

**int** L[30]; **//Количество элементов в массиве L**

**//совпадает с количеством вершин графа.**

**int** num=0;

**class** Spisok

{

**private**:

Lref Head; **//Указатель на голову списка заголовочных узлов.**

Lref Tail; **//Указатель на фиктивный элемент**

**//в конце списка заголовочных узлов.**

svqz Stack; **//Указатель на рабочий стек.**

**void** SearchGraph (int, Lref \*);

**void** W\_S (svqz \*, TipElement, TipElement);

**void** UDALENIE (svqz \*, TipElement \*, TipElement \*);

**public**:

Spisok() {**//Инициализация списка заголовочных узлов.**

Head = Tail = **new** (Leader); Stack = **NULL**;}

Lref GetHead() { **return** Head; }

Lref &GetHead1() { **return** Head; }

Lref GetTail() { **return** Tail; }

**void** MakeGraph ();

**void** PrintGraph ();

**void** TwoLink (Lref, Lref);

};

**void** main ()

{

Spisok A;

Lref SuperHead;**//Заглавное звено списка заголовочных узлов.**

Lref t; **//Рабочий указатель для перемещения**

**//по списку заголовочных звеньев.**

**//Построение графа и вывод его структуры Вирта.**

A.MakeGraph ();

A.PrintGraph (); cout<<endl;

**//Нахождение компонент двусвязности графа.**

cout << "Множество ребер всех компонент двусвязности:\n";

t = A.GetHead();

**while** (t!=A.GetTail())

{ (\*t).WGN = 0; t = (\*t).Next;}

**//Построение заглавного звена списка заголовочных звеньев.**

SuperHead = **new** (Leader);

SuperHead->Key = 0; SuperHead->Next = A.GetHead();

A.GetHead1() = SuperHead;

t = A.GetHead()->Next;

**while** (t!=A.GetTail())

{

**if** ( t->WGN==0 ) A.TwoLink (t,A.GetHead());

t = t->Next;

}

}

**void** Spisok::SearchGraph (**int** w, Lref \*h)

**//Функция возвращает указатель на заголовочный узел**

**//с ключом w в графе, заданном структурой Вирта с указателем Head.**

{

\*h = Head; (\*Tail).Key = w;

**while** ((\*\*h).Key!=w) \*h = (\*\*h).Next;

**if** (\*h==Tail)

**//В списке заголовочных узлов нет узла с ключом w.**

**//Поместим его в конец списка Head.**

{ Tail = **new** (Leader); (\*\*h).Count = 0;

(\*\*h).Trail = **NULL**; (\*\*h).Next = Tail; }

}

**void** Spisok::MakeGraph ()

**//Функция возвращает указатель Head на структуру**

**//Вирта, соответствующую ориентированному графу.**

{

**int** x,y;

Lref p,q; **//Рабочие указатели.**

Tref t,r; **//Рабочие указатели.**

Boolean Res; **//Флаг наличия дуги.**

cout<<"Вводите начальную вершину дуги: ";

cin>>x;

**while** (x!=0)

{

cout<<"Вводите конечную вершину дуги: "; cin>>y;

**//Определим, существует ли в графе дуга (x,y)?**

SearchGraph (x, &p); SearchGraph (y,&q);

r = (\*p).Trail; Res = FALSE;

**while** ((r!=**NULL**)&&(!Res))

**if** ((\*r).Id==q) Res = TRUE;

**else** r = (\*r).Next;

**if** (!Res) **//Если дуга отсутствует, то поместим её в граф.**

{ t = **new** (Trailer); (\*t).Id = q;

(\*t).Next = (\*p).Trail; (\*p).Trail = t; (\*q).Count++; }

cout<<"Вводите начальную вершину дуги: "; cin>>x;

}

}

**void** Spisok::PrintGraph ()

**//Вывод структуры Вирта, заданной указателем**

**//Head и соответствующей ориентированному графу.**

{

Lref p; **//Рабочий указатель.**

Tref q; **//Рабочий указатель.**

p = Head;

**while** (p!=Tail)

{

cout<<"("<<(\*p).Key; q = (\*p).Trail;

**while** (q!=**NULL**)

{ cout<<(\*(\*q).Id).Key; q = (\*q).Next; }

cout<<")"; p = (\*p).Next; cout<<" ";

}

}

**void** Spisok::W\_S (svqz \*stk, TipElement el1, TipElement el2)

**//Помещение el1 и el2 в стек stk.**

{

svqz q=new (St);

(\*q).Element1 = el1; (\*q).Element2 = el2;

(\*q).Sled = \*stk; \*stk = q;

}

**void** Spisok::UDALENIE (svqz \*stk, TipElement \*klad1, TipElement \*klad2)

**//Удаление звена из стека, заданного указателем \*stk.**

**//Значение информационного поля удаляемого звена сохраня-**

**//ется в параметрах klad1 и klad2.**

{

svqz q;

**if** (\*stk==**NULL**) cout<<"Попытка выбора из пустого стека!\n";

**else**

{ \*klad1 = (\*\*stk).Element1;

\*klad2 = (\*\*stk).Element2;

q = \*stk; \*stk = (\*\*stk).Sled; **delete** q; }

}

**void** Spisok::TwoLink (Lref r, Lref p)

**//Поиск в глубину, начиная с вершины r->Key и полагая,**

**//что p является указателем на отца вершины r->Key.**

{

Tref t;

Lref e1,e2;

num++; r->WGN = num; L[r->Key] = r->WGN;

t = r->Trail;

**while** ( t != **NULL** )

{

**if** ( t->Id->WGN==0 )

{

W\_S (&Stack,r,t->Id);

TwoLink (t->Id,r);

**if** ( L[r->Key] > L[t->Id->Key] )

L[r->Key] = L[t->Id->Key];

**if** ( L[t->Id->Key] >= r->WGN )

{

**//Выписать ребра компоненты двусвязности.**

**do** {

UDALENIE (&Stack,&e1,&e2);

cout << "(" << e1->Key << " " << e2->Key << ") ";

}

**while** (!((e1->Key==r->Key && e2->Key==t->Id->Key) ||

(e1->Key==t->Id->Key && e2->Key==r->Key)));

cout << ";\n";

}

}

**else**

**if** (t->Id->Key != p->Key && r->WGN > t->Id->WGN)

{

W\_S (&Stack,r,t->Id);

**if** ( L[r->Key] > t->Id->WGN )

L[r->Key] = t->Id->WGN;

}

t = t->Next;

}

}

Текст этой программы можно взять [здесь](http://khpi-iip.mipk.kharkiv.edu/library/datastr/book_sod/kgsu/din106_1.zip).

Результат работы приложения изображен на рисунке 1:

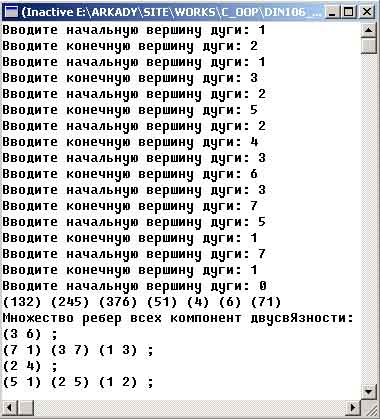


Рис.1. Результат работы приложения

(1) Липский В. Комбинаторика для программистов: Пер. с польск. - М.: Мир, 1988. - 213 с.

Со следующего шага мы начнем знакомиться с ***остовами***.

## Остовы. Построение остова

На этом шаге мы рассмотрим ***алгоритм построения остова***.

***Определение.***

Для произвольного связного неориентированного графа **G=(V,E)** каждое дерево **(V,T)**, где **T** - подмножество **E**, будем называть ***стягивающим деревом (остовом, остовным деревом)*** графа **G**. Ребра такого дерева будем называть ***ветвями***, а все остальные ребра графа будем называть ***хордами***.

Максимальный подграф без циклов произвольного графа **G** называется ***стягивающим лесом*** графа **G**.

Задачу нахождения стягивающих деревьев можно понимать как поиск экономных путей, обеспечивающих связь между точками заданного множества без ненужного дублирования. Поэтому она применима во многих областях, например, при исследовании электрических цепей или при анализе схем программ.

***Построение остова***

Процедуры обхода графа в глубину и в ширину можно простым способом использовать для нахождения ***остовов***. В обоих случаях достижение новой вершины графа **u** из вершины **v** вызывает "включение" в остовное дерево ребра **{u,v}**.

Сравните две процедуры:

**void** Spisok::Depth\_First\_Search (Lref r)

**//Рекурсивный обход графа в глубину. r - указатель**

**//на структуру Вирта.**

{

Tref t;

t = (\*r).Trail; cout<<(\*r).Key; (\*r).Flag = FALSE;

**while** (t!=**NULL**)

{ **if** ((\*(\*t).Id).Flag) Depth\_First\_Search ((\*t).Id); t = (\*t).Next; }

}

**void** Spisok::Ostov\_Depth (Lref r)

**//Рекурсивный обход графа в глубину, соединенный с**

**//нахождением ребра остовного дерева.**

**//r - указатель на структуру Вирта.**

{

Tref t;

Lref s;

s = r;

t = (\*r).Trail; (\*r).Flag = FALSE;

**while** ( t != **NULL** )

{ **if** ((\*(\*t).Id).Flag)

{ cout << "(" << s->Key << "," << t->Id->Key << ") ";

Ostov\_Depth ((\*t).Id);

}

t = (\*t).Next;

}

}

Для доказательства того, что функция **Ostov\_Depth** корректно строит остов связного графа, достаточно отметить следующие три факта [1, с.95]:

1) в момент выдачи на экран новой ветви **{s->Key,t->Id->Key}** оператором

cout << "(" << s->Key << "," << t->Id->Key << ") ";

в дереве уже существует путь из вершины **Head->Key** в **s->Key**, что легко доказывается по индукции. Таким образом, процедура строит связный граф;

2) каждая ***новая ветвь***, добавляемая к множеству **T** (конечно, в процедуре множества нет: его роль играет экран дисплея!), соединяет ***уже рассмотренную*** вершину **s->Key** c ***новой*** вершиной **t->Id->Key**. Отсюда следует, что построенный граф не содержит циклов: действительно, последняя ветвь, "замыкающая" цикл, должна была бы соединить две уже рассмотренные вершины;

3) из свойства функции **Depth\_First\_Search** поиска в глубину следует, что функция **Ostov\_Depth** просматривает все вершины связного графа.

Следовательно, граф **(V,T)**, построенный процедурой **Ostov\_Depth**, ***есть стягивающее дерево графа* G**.

***Вычислительная сложность алгоритма*** есть, очевидно, **O(n+m)**, т.е. того же порядка, что и поиск в глубину. Здесь: **n** - число вершин, а **m** - число ребер графа.

Приведем демонстрационный пример.

Пример 1. ***Нахождение стягивающего дерева связного графа, используя рекурсивный обход графа в глубину***. Граф задан структурой Вирта.

**#include** <iostream.h>

**#define** TRUE 1

**#define** FALSE 0

**typedef** **int** Boolean;

**typedef** **struct** L \*Lref; **// Тип: указатель на заголовочный узел.**

**typedef** **struct** T \*Tref; **// Тип: указатель на дуговой узел.**

**//Описание типа заголовочного узла.**

**typedef** **struct** L

{

**int** Key; **//Имя заголовочного узла.**

**int** Count; **//Количество предшественников.**

Boolean Flag; **//Флаг посещения узла при обходе.**

Tref Trail; **//Указатель на список смежности.**

Lref Next; **//Указатель на следующий узел в списке заголовочных узлов.**

} Leader;

**//Описание типа дугового узла.**

**typedef** **struct** T

{

Lref Id;

Tref Next;

} Trailer;

**//Описание типа узла стека.**

**typedef** Tref TipElement;

**typedef** **struct** Zveno \*svqz;

**typedef** **struct** Zveno

{

TipElement Element; **//Указатель на список смежности.**

svqz Sled;

} St;

**class** Spisok

{

**private**:

Lref Head; **//Указатель на голову списка заголовочных узлов.**

Lref Tail; **//Указатель на фиктивный элемент**

**// в конце списка заголовочных узлов.**

**void** SearchGraph (int, Lref \*);

**public**:

Spisok() {**//Инициализация списка заголовочных узлов.**

Head = Tail = **new** (Leader); }

Lref GetHead() { **return** Head; }

Lref GetTail() { **return** Tail; }

**void** MakeGraph ();

**void** Ostov\_Depth (Lref);

**void** PrintGraph ();

};

**void** main ()

{

Spisok A;

Lref t; **//Рабочий указатель для перемещения**

**// по списку заголовочных звеньев.**

**//Построение графа и вывод его структуры Вирта.**

A.MakeGraph ();

A.PrintGraph (); cout<<endl;

**//Построение стягивающего дерева (остова).**

cout<<"Остов: ";

t = A.GetHead();

**while** (t!=A.GetTail())

{ (\*t).Flag = TRUE; t = (\*t).Next; }

A.Ostov\_Depth (A.GetHead()); cout<<endl;

}

**void** Spisok::SearchGraph (**int** w, Lref \*h)

**//Функция возвращает указатель на заголовочный узел**

**//с ключом w в графе, заданном структурой Вирта с указателем Head.**

{

\*h = Head; (\*Tail).Key = w;

**while** ((\*\*h).Key!=w) \*h = (\*\*h).Next;

**if** (\*h==Tail)

**//В списке заголовочных узлов нет узла с ключом w.**

**//Поместим его в конец списка Head.**

{ Tail = **new** (Leader); (\*\*h).Count = 0;

(\*\*h).Trail = **NULL**; (\*\*h).Next = Tail; }

}

**void** Spisok::MakeGraph ()

**//Функция возвращает указатель Head на структуру**

**//Вирта, соответствующую ориентированному графу.**

{

**int** x,y;

Lref p,q; **//Рабочие указатели.**

Tref t,r; **//Рабочие указатели.**

Boolean Res; **//Флаг наличия дуги.**

cout<<"Вводите начальную вершину дуги: ";

cin>>x;

**while** (x!=0)

{

cout<<"Вводите конечную вершину дуги: "; cin>>y;

**//Определим, существует ли в графе дуга (x,y)?**

SearchGraph (x, &p); SearchGraph (y,&q);

r = (\*p).Trail; Res = FALSE;

**while** (( r != **NULL** )&&(!Res))

**if** ((\*r).Id==q) Res = TRUE;

**else** r = (\*r).Next;

**if** (!Res) **//Если дуга отсутствует, то поместим её в граф.**

{ t = **new** (Trailer); (\*t).Id = q;

(\*t).Next = (\*p).Trail; (\*p).Trail = t; (\*q).Count++; }

cout<<"Вводите начальную вершину дуги: "; cin>>x;

}

}

**void** Spisok::PrintGraph ()

**//Вывод структуры Вирта, заданной указателем**

**//Head и соответствующей ориентированному графу.**

{

Lref p; **//Рабочий указатель.**

Tref q; **//Рабочий указатель.**

p = Head;

**while** (p!=Tail)

{

cout<<"("<<(\*p).Key; q = (\*p).Trail;

**while** (q!=**NULL**)

{ cout<<(\*(\*q).Id).Key; q = (\*q).Next; }

cout<<")"; p = (\*p).Next; cout<<" ";

}

}

**void** Spisok::Ostov\_Depth (Lref r)

**//Рекурсивный обход графа в глубину, соединенный с**

**//нахождением ребра остовного дерева.**

**//r - указатель на структуру Вирта.**

{

Tref t;

Lref s;

s = r;

t = (\*r).Trail; (\*r).Flag = FALSE;

**while** ( t != **NULL** )

{ **if** ((\*(\*t).Id).Flag)

{ cout << "(" << s->Key << "," << t->Id->Key << ") ";

Ostov\_Depth ((\*t).Id);

}

t = (\*t).Next;

}

}

Текст этой программы можно взять [здесь](http://khpi-iip.mipk.kharkiv.edu/library/datastr/book_sod/kgsu/din107_1.zip).

Результат работы приложения изображен на рисунке 1:

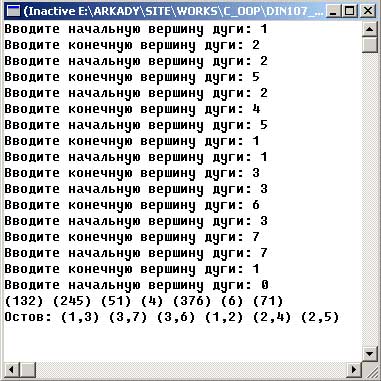


Рис.1. Результат работы приложения

Ясно, что далеко не всякий ***ориентированный*** граф имеет остов. Например, граф, заданный следующим списком дуг: (1,2), (1,6), (1,7), (6,7), (7,2), (5,7), (2,5), (3,2), (3,5) остова не имеет.

Подобным же способом можно построить остов, используя обход графа в ширину. В функции **Ostov\_Breadth**используются ***две очереди***.

Пример 2. ***Нахождение стягивающего дерева связного графа с использованием нерекурсивного обхода графа в ширину***. Граф задан структурой Вирта.

**#include** <iostream.h>

**#define** TRUE 1

**#define** FALSE 0

**typedef** **int** Boolean;

**typedef** **struct** Leader \*Lref; **// Тип: указатель на заголовочный узел.**

**typedef** **struct** Trailer \*Tref; **// Тип: указатель на дуговой узел.**

**//Описание типа заголовочного узла.**

**typedef** **struct** Leader

{

**int** Key; **//Имя заголовочного узла.**

**int** Count; **//Количество предшественников.**

Boolean Flag; **//Флаг посещения узла при обходе.**

Tref Trail; **//Указатель на список смежности.**

Lref Next; **//Указатель на следующий узел в списке заголовочных узлов.**

};

**//Описание типа дугового узла.**

**typedef** **struct** Trailer

{

Lref Id;

Tref Next;

};

**//Описание типа узла очереди.**

**typedef** Tref TipElement; **//Указатель на звено заголовочного списка.**

**typedef** **struct** Zveno \*svqz;

**typedef** **struct** Zveno

{

TipElement Element; **//Указатель на список смежности.**

svqz Sled;

};

**//Описание типа узла очереди указателей на**

**//узлы списков смежности.**

**typedef** Lref TipElement1;

**typedef** **struct** Zveno1 \*svqz1;

**typedef** **struct** Zveno1

{

TipElement1 Element; **//Указатель на заголовочный узел.**

svqz1 Sled;

};

**class** Spisok

{

**private**:

Lref Head; **//Указатель на голову списка заголовочных узлов.**

Lref Tail; **//Указатель на фиктивный элемент**

**// в конце списка заголовочных узлов.**

**void** Udalenie\_A (svqz \*, svqz \*, TipElement \*);

**void** Udalenie\_A1 (svqz1 \*, svqz1 \*, Lref \*);

**void** Dobavlenie (svqz \*, svqz \*, TipElement);

**void** Dobavlenie1 (svqz1 \*, svqz1 \*, Lref);

**void** SearchGraph (int, Lref \*);

**public**:

Spisok() {**//Инициализация списка заголовочных узлов.**

Head = Tail = **new** (Leader); }

Lref GetHead() { **return** Head; }

Lref GetTail() { **return** Tail; }

**void** MakeGraph ();

**void** PrintGraph ();

**void** Ostov\_Breadth ();

};

**void** main ()

{

Spisok A;

Lref t; **//Рабочий указатель для перемещения**

**//по списку заголовочных звеньев.**

**//Построение графа и вывод его структуры Вирта.**

A.MakeGraph ();

A.PrintGraph (); cout<<endl;

**//Построение стягивающего дерева (остова).**

cout<<"Остов: ";

t = A.GetHead();

**while** (t!=A.GetTail())

{ (\*t).Flag = TRUE; t = (\*t).Next; }

A.Ostov\_Breadth (); cout<<endl;

}

**void** Spisok::SearchGraph (**int** w, Lref \*h)

**//Функция возвращает указатель на заголовочный узел**

**//с ключом w в графе, заданном структурой Вирта с указателем Head.**

{

\*h = Head; (\*Tail).Key = w;

**while** ((\*\*h).Key!=w) \*h = (\*\*h).Next;

**if** (\*h==Tail)

**//В списке заголовочных узлов нет узла с ключом w.**

**//Поместим его в конец списка Head.**

{ Tail = **new** (Leader); (\*\*h).Count = 0;

(\*\*h).Trail = **NULL**; (\*\*h).Next = Tail; }

}

**void** Spisok::MakeGraph ()

**//Функция возвращает указатель Head на структуру**

**//Вирта, соответствующую ориентированному графу.**

{

**int** x,y;

Lref p,q; **//Рабочие указатели.**

Tref t,r; **//Рабочие указатели.**

Boolean Res; **//Флаг наличия дуги.**

cout<<"Вводите начальную вершину дуги: ";

cin>>x;

**while** (x!=0)

{

cout<<"Вводите конечную вершину дуги: "; cin>>y;

**//Определим, существует ли в графе дуга (x,y)?**

SearchGraph (x, &p); SearchGraph (y,&q);

r = (\*p).Trail; Res = FALSE;

**while** ((r!=**NULL**)&&(!Res))

**if** ((\*r).Id==q) Res = TRUE;

**else** r = (\*r).Next;

**if** (!Res) **//Если дуга отсутствует, то поместим её в граф.**

{ t = **new** (Trailer); (\*t).Id = q;

(\*t).Next = (\*p).Trail; (\*p).Trail = t; (\*q).Count++; }

cout<<"Вводите начальную вершину дуги: "; cin>>x;

}

}

**void** Spisok::PrintGraph ()

**//Вывод структуры Вирта, заданной указателем**

**//Head и соответствующей ориентированному графу.**

{

Lref p; **//Рабочий указатель.**

Tref q; **//Рабочий указатель.**

p = Head;

**while** (p!=Tail)

{

cout<<(\*p).Key<<"("; q = (\*p).Trail;

**while** (q!=**NULL**)

{ cout<<(\*(\*q).Id).Key; q = (\*q).Next; }

cout<<")"; p = (\*p).Next; cout<<" ";

}

}

**void** Spisok::Dobavlenie (svqz \*L, svqz \*R, TipElement elem)

**//Добавление элемента elem в очередь, заданную указателями L и R.**

{

svqz K = **new** (Zveno);

K->Element = elem; K->Sled = **NULL**;

**if** (\*L==**NULL**)

{ (\*L) = K; (\*R) = K; }

**else** { (\*R)->Sled = K; (\*R) = K; }

}

**void** Spisok::Dobavlenie1 (svqz1 \*L, svqz1 \*R, Lref elem)

**//Добавление элемента elem в очередь, заданную указателями L и R.**

{

svqz1 K = **new** (Zveno1);

K->Element = elem; K->Sled = **NULL**;

**if** (\*L==**NULL**)

{ (\*L) = K; (\*R) = K; }

**else** { (\*R)->Sled = K; (\*R) = K; }

}

**void** Spisok::Udalenie\_A (svqz \*L, svqz \*R, TipElement \*A)

**//Удаление элемента из очереди, заданной указателями L и R и**

**//помещение удаленного элемента в переменную A.**

{

svqz q;

**if** ((\*L)!=**NULL**)

**if** ((\*L)->Sled!=**NULL**)

{

(\*A) = (\*L)->Element; q = (\*L);

(\*L) = (\*L)->Sled; **delete** q;

}

**else** {

(\*A) = (\*L)->Element; **delete** \*L;

(\*L) = (\*R) = **NULL**;

}

}

**void** Spisok::Udalenie\_A1 (svqz1 \*L, svqz1 \*R, Lref \*A)

**//Удаление элемента из очереди, заданной указателями L и R и**

**//помещение удаленного элемента в переменную A.**

{

svqz1 q;

**if** ((\*L)!=**NULL**)

**if** ((\*L)->Sled!=**NULL**)

{

(\*A) = (\*L)->Element; q = (\*L);

(\*L) = (\*L)->Sled; **delete** q;

}

**else** {

(\*A) = (\*L)->Element; **delete** \*L;

(\*L) = (\*R) = **NULL**;

}

}

**void** Spisok::Ostov\_Breadth ()

**//Обход графа, заданного указателем r в ширину (нерекурсивный**

**//обход), соединенный с построением стягивающего дерева.**

{

svqz L; **//Указатель на начало очереди.**

svqz R; **//Указатель на конец очереди.**

svqz1 L1; **//Указатель на начало очереди заглавных узлов.**

svqz1 R1; **//Указатель на конец очереди заглавных узлов.**

Tref s; **//Рабочий указатель.**

Tref t; **//Рабочий указатель.**

Tref Tail1;**//Указатель на фиктивный элемент в очереди L,R;**

Lref v; **//Рабочий указатель.**

Tail1 = **new** (Trailer); **//Построили фиктивный элемент.**

**//Создали пустые очеpеди.**

L = R = **NULL**;

L1 = R1 = **NULL**;

**//Посетим первую непосещенную вершину графа.**

Head->Flag = FALSE;

t = Head->Trail;

**while** ( t != **NULL** )

{ Dobavlenie (&L,&R,t); t = t->Next; }

Dobavlenie (&L,&R,Tail1);

v = Head;

**while** ( L!=**NULL** )

**//Пока очередь не пуста...**

{

Udalenie\_A (&L,&R,&t);

**if** ( t != Tail1 )

{

**if** (t->Id->Flag)

{

cout << "(" << v->Key << "," << t->Id->Key << ") ";

t->Id->Flag = FALSE;

}

s = t->Id->Trail;

Dobavlenie1 (&L1,&R1,t->Id);

**while** (s != **NULL** )

{

**if** (s->Id->Flag)

{

Dobavlenie (&L,&R,s);

t->Id->Flag = FALSE;

}

s = s->Next;

}

Dobavlenie (&L,&R,Tail1);

}

**else** Udalenie\_A1 (&L1,&R1,&v);

}

}

Текст этой программы можно взять [здесь](http://khpi-iip.mipk.kharkiv.edu/library/datastr/book_sod/kgsu/din107_2.zip).

Результат работы приложения изображен на рисунке 2:

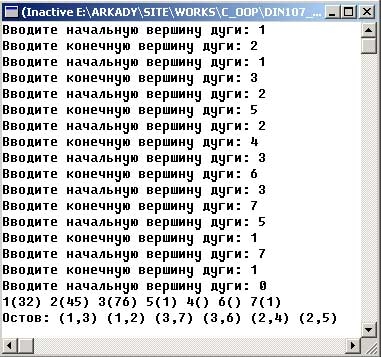


Рис.2. Результат работы приложения

***Замечание.*** *Приведенные программы легко модифицируются для произвольных графов,* ***не обязательно связных****.*

Обратимся теперь к реализации алгоритма построения ***остовного дерева связного графа*** на языке **LISP**. Для этого вспомним, что при [поиске в глубину](http://khpi-iip.mipk.kharkiv.edu/library/datastr/book_sod/kgsu/din_0096.html#1) новая вершина добавляется к списку уже пройденных вершин только в том случае, когда она в этом списке отсутствует. Поэтому последовательность вершин, соответствующая удлинению списка **PATH** при работе функции **DEFI**, представляет собой ***простой***0(т.е. без самопересечений) ***путь***в просматриваемом графе.

После этих замечаний можно написать функцию, значением которой является список ребер, представляющий ***остовное дерево графа***:

Программа 1 [2, с.130-131].

(DEFUN SPANDF (LAMBDA (GRAPH ROOT)

**; Построение остовного дерева графа GRAPH с корнем ROOT:**

**; GRAPH - граф, представленный структурой смежности в**

**; виде ассоциативного списка,**

**; ROOT - корень остовного дерева,**

**; Результат: список ребер остовного дерева, полученного**

**; с помощью поиска в глубину**

(COND ( (**NULL** GRAPH) NIL )

( T (SPDF GRAPH (LIST ROOT) (LIST ROOT)) )

)

))

**; --------------------------------------**

(DEFUN SPDF (LAMBDA (GRAPH VISITED PATH)

**; GRAPH - граф, представленный структурой смежности в**

**; виде ассоциативного списка,**

**; VISITED - список уже посещенных вершин,**

**; PATH - список вершин, определяющий путь посещения**

(COND ( (**NULL** PATH) NIL )

( T (COND ( (**NULL** (EXPND GRAPH VISITED (CAR PATH)))

(SPDF GRAPH VISITED (CDR PATH)) )

( T (CONS

(CONS (CAR PATH)

(EXPND GRAPH VISITED

(CAR PATH)))

(SPDF

GRAPH

(CONS (EXPND GRAPH

VISITED

(CAR PATH))

VISITED)

(CONS (EXPND GRAPH

VISITED

(CAR PATH))

PATH))) )) )

)

))

**; -----------------------------------------**

(DEFUN EXPND (LAMBDA (GRAPH VISITED VERTEX)

**; Выбор в графе GRAPH следующей еще не просмотренной**

**; вершины, смежной с вершиной VERTEX**

(COND ( (**NULL** (NEIGHBOUR3 VERTEX GRAPH)) NIL )

( T (FIRSTNOTVISITED

VISITED

(NEIGHBOUR3 VERTEX GRAPH)) ))

))

**; --------------------------------------------**

(DEFUN FIRSTNOTVISITED (LAMBDA (VISITED VLIST)

**; Поиск первой непосещенной вершины в списке VLIST**

**; VISITED - список уже посещенных вершин**

(COND ( (**NULL** VLIST) NIL )

( T (COND ( (**NULL** (MEMBER (CAR VLIST) VISITED))

(CAR VLIST) )

( T (FIRSTNOTVISITED

VISITED

(CDR VLIST)) )) ))

))

**; ---------------------------------**

(DEFUN NEIGHBOUR3 (LAMBDA (X GRAPH)

**; Функция возвращает список вершин графа GRAPH, смежных с**

**; вершиной X**

(COND ( (**NULL** (ASSOC X GRAPH)) NIL )

( T (CDR (ASSOC X GRAPH)) ))

))

Текст этой библиотеки можно взять [здесь](http://khpi-iip.mipk.kharkiv.edu/library/datastr/book_sod/kgsu/din107_3.zip).

Тестовые примеры:

1) $ (SPANDF '((1 . (2 3)) (2 . (4 5)) (3 . (6 7))) 1)

((1 . 2) (2 . 4) (2 . 5) (1 . 3) (3 . 6) (3 . 7))

2) $ (SETQ A1 '((1 . (2 4)) (2 . (1 3 5)) (3 . (2 4 6))

(4 . (1 3 6 7)) (5 . (2 6 8 9)))

((1 2 4) (2 1 3 5) (3 2 4 6) (4 1 3 6 7) (5 2 6 8 9))

$ (SETQ A2 '((6 . (3 4 5 7 9)) (7 . (4 6)) (8 . (5)) (9 . (5 6)))

((6 3 4 5 7 9) (7 4 6) (8 5) (9 5 6))

$ (SPANDF (APPEND A1 A2) 1)

((1 . 2) (2 . 3) (3 . 4) (4 . 6) (6 . 5) (5 . 8) (5 . 9) (6 . 7))

Теперь мы зададимся вопросом, какими свойствами будет обладать остовное дерево, построенное с помощью процедуры [***обхода графа в ширину***](http://khpi-iip.mipk.kharkiv.edu/library/datastr/book_sod/kgsu/din_0097.html#1)? Мы уже упоминали о том, что при обходе в ширину просмотренные вершины располагаются в порядке неубывания их расстояния от начальной вершины. Поэтому в остовном дереве, построенном с помощью обхода в ширину, единственный путь, соединяющий какую-либо вершину с начальной, совпадает с кратчайшим путем в исходном графе между этими двумя вершинами.

Необходимые для построения остовного дерева с помощью обхода в ширину изменения в функции **BRFI** более существенны по сравнению с аналогичными изменениями в функции **DEFI**. Дело в том, что при поиске в ширину нам не нужна была информация о предшественнике рассматриваемой вершины - в этом смысле поиск в ширину нелокален. Поэтому вместо очереди вершин, предназначенных для просмотра, мы введем ***очередь***, состоящую непосредственно из элементов структуры смежности.

Программа 2 [3, с.132-133].

(DEFUN SPANBF (LAMBDA (GRAPH ROOT)

**; Построение остовного дерева графа GRAPH с корнем**

**; в вершине ROOT при помощи обхода графа в ширину:**

**; GRAPH - граф, представленный структурой смежности в**

**; виде ассоциативного списка,**

**; ROOT - корень остовного дерева**

(SPBF GRAPH (LIST ROOT) (CDR (ASSOC ROOT GRAPH))

NIL (CAR (ASSOC ROOT GRAPH)))

))

**; ---------------------------------------------------**

(DEFUN SPBF (LAMBDA (GRAPH VISITED HEAD QUEUE FATHER)

**; GRAPH - граф, представленный структурой смежности в**

**; виде ассоциативного списка,**

**; VISITED - список уже посещенных вершин,**

**; HEAD - список вершин для просмотра,**

**; QUEUE - очередь списка вершин для просмотра,**

**; FATHER - предок просматриваемых вершин**

(COND ( (**NULL** HEAD)

(COND ( (**NULL** QUEUE) NIL )

( T (SPBF GRAPH VISITED

(CDAR QUEUE) (CDR QUEUE)

(CAAR QUEUE)) )) )

( T (COND ( (MEMBER (CAR HEAD) VISITED)

(SPBF GRAPH VISITED (CDR HEAD) QUEUE

FATHER) )

( T (CONS

(CONS FATHER (CAR HEAD))

(SPBF

GRAPH

(CONS (CAR HEAD) VISITED)

(CDR HEAD)

(APPEND QUEUE

(LIST (ASSOC (CAR HEAD)

GRAPH)))

FATHER)) )) )

)

))

Текст этой библиотеки можно взять [здесь](http://khpi-iip.mipk.kharkiv.edu/library/datastr/book_sod/kgsu/din107_4.zip).

Тестовые примеры:

1) $ (SPANBF '((1 . (2 4)) (2 . (1 3)) (4 . (1 3)) (3 . (1 4))) 1)

((1 . 2) (1 . 4) (2 . 3))

$ (SPANBF '((1 . (2 4)) (2 . (1 3)) (4 . (1 3)) (3 . (1 4))) 2)

((2 . 1) (2 . 3) (1 . 4))

$ (SPANBF '((1 . (2 4)) (2 . (1 3)) (4 . (1 3)) (3 . (1 4))) 3)

((3 . 1) (3 . 4) (1 . 2))

$ (SPANBF '((1 . (2 4)) (2 . (1 3)) (4 . (1 3)) (3 . (1 4))) 4)

((4 . 1) (4 . 3) (1 . 2))

2) $ (SPANBF '((1 . (2 3)) (2 . (4 5)) (3 . (6 7))) 1)

((1 . 2) (1 . 3) (2 . 4) (2 . 5) (3 . 6) (3 . 7))

3) $ (SETQ A1 '((1 . (2 4)) (2 . (1 3 5)) (3 . (2 4 6))

(4 . (1 3 6 7)) (5 . (2 6 8 9)))

((1 2 4) (2 1 3 5) (3 2 4 6) (4 1 3 6 7) (5 2 6 8 9))

$ (SETQ A2 '((6 . (3 4 5 7 9)) (7 . (4 6)) (8 . (5)) (9 . (5 6)))

((6 3 4 5 7 9) (7 4 6) (8 5) (9 5 6))

$ (SPANBF (APPEND A1 A2) 1)

((1 . 2) (1 . 4) (2 . 3) (2 . 5) (4 . 6) (4 . 7) (5 . 8) (5 . 9))

Обратите внимание, что при работе функции **SPBF** параметр **HEAD** всегда есть список соседей вершины, определяемой параметром **FATHER**, что позволяет знать предшественника текущей вершины.

(1) Липский В. Комбинаторика для программистов. - М.: Мир, 1988. - 213 с.

(2) Turbo Pascal Version 3.0 Reference Manual. Borland International, 1985.

(3) Крюков А.П., Радионов А.Я., Таранов А.Ю., Шаблыгин Е.М. Программирование на языке R-Лисп. - М.: Радио и связь, 1991. - 192 с.

На следующем шаге мы рассмотрим ***алгоритм построения остова наименьшей стоимости***.

## Остовы. Построение остова наименьшей стоимости

На этом шаге мы рассмотрим ***алгоритм построения остова наименьшей стоимости***.

Пусть **G=(V,E)** - связный неориентированный граф, для которого задана ***функция стоимости***, отображающая ребра в вещественные (целые) числа. Напомним, что ***остовным деревом*** для данного графа называется неориентированное дерево, содержащее все вершины графа. ***Стоимость остовного дерева*** определяется как сумма стоимостей его ребер.

Наша ***цель*** - найти для **G** остовное дерево наименьшей стоимости.

Эта задача возникает при проектировании линий электропередачи, трубопроводов, дорог и т.п., когда требуется заданные центры соединить некоторой системой каналов связи так, чтобы любые два центра были связаны либо непосредственно соединяющим их каналом, либо через другие центры и каналы, и чтобы общая длина (или, например, стоимость) каналов связи была минимальной. В этой ситуации заданные центры можно считать вершинами графа с весами ребер, равными длинам (стоимости) соединяющих эти центры каналов. Тогда искомая сеть будет остовным деревом наименьшей длины (стоимости). Поскольку граф с **n** вершинами содержит **nn-2** различных остовных дерева, то решение этой задачи "слепым" перебором" вариантов потребовало бы чрезвычайно больших вычислений даже при относительно малых **n**. Однако для ее решения имеются эффективные алгоритмы.

Первый алгоритм нахождения такого дерева - ***алгоритм Дж.Краскала***, опубликованный в 1956 г. Развитием этого алгоритма является алгоритм ***Р.Прима*** (1957 г.) (см. Прим Р.К. Кратчайшие связывающие сети и их обощения. - Киберн.сб., вып.2 (ст.серия), 1961, с.95-107).

Приведем теорему, позволяющую получить ***алгоритм Краскала***.

***Теорема [1, с.69].***

Пусть **G** - связный граф с **n** вершинами, и пусть каждому его ребру приписано неотрицательное действительное число **m(e)**, называемое его ***мерой***.

Тогда следующая процедура приводит к решению задачи о минимальном остовном графе:

* выберем ребро **e1**, обладающее в **G** наименьшей мерой;
* определим по индукции последовательность ребер **e2,e3, ...,en-1**, выбирая на каждом шаге ребро (отличное от предыдущих) с наименьшей мерой, обладающее тем свойством, что оно не образует циклов с предыдущими ребрами **ei**. Полученный подграф **T** графа **G**, ребрами которого являются **e2,e3, ...,en-1** и есть требуемое остовное дерево.

Алгоритм Краскала [2, с.199] работает с набором **VS** непересекающихся множеств узлов. Каждое множество **W** из **VS** представляет связное множество узлов, образующее остовное дерево в остовном лесу, представленном набором **VS**.

Ребра выбираются из **E** в порядке возрастания стоимости. Ребра **(v,w)** рассматриваются по очереди. Если **v** и **w**принадлежат одному и тому же множеству из **VS**, то ребро **(v,w)** исключается из рассмотрения. Если **v** и **w**принадлежат разным множествам **W1** и **W2** (это означает, что **W1** и **W2** еще не соединены), то сливаем их в одно множество и добавляем **(v,w)** к множеству ребер, входящих в окончательное остовное дерево.

В [2, с.199] приведено описание алгоритма Краскала на псевдокоде:

T <- пустое множкство;

VS <- пустое множество;

Построить очередь с приоритетами Q, содержащую все ребра из E;

**for** (v принадлежащего V) Добавить {v} к VS

**while** |VS|>1

{

Выбрать в Q ребро (v,w) наименьшей стоимости;

Удалить (v,w) из Q;

**if** v и w принадлежат различным множествам W1 и W2 из VS

{

Заменить W1 и W2 на объединение W1 и W2 в VS;

Добавить (v,w) к T;

}

}

Реализуем алгоритм на языке **C++**. Для хранения ***ребер и их стоимостей*** воспользуемся ***бинарным деревом поиска***, описание которого имеет вид:

**typedef** **unsigned** **int** SubInt;

**typedef** **struct** Uzel \*Ref;

**typedef** **struct** Uzel

{

SubInt X; **//Начало дуги.**

SubInt Y; **//Конец дуги**

**int** Pay; **//Стоимость дуги.**

Ref Left; **//Указатель на левого сына.**

Ref Right;**//Указатель на правого сына.**

};

Ясно, что в "самой левой" вершине дерева будет храниться дуга, обладающая наименьшей стоимостью, и ***поиск этой вершины*** может выглядеть, например, так:

UkUzel = Root; **//Установили текущий указатель на корень дерева.**

**while** (UkUzel->Left != **NULL**)

UkUzel = UkUzel->Left;

T1 = UkUzel->X; T2 = UkUzel->Y; **//Получили ребро!**

Множество **VS** будем хранить в ***линейном однонаправленном списке с заглавным звеном***, описание которого выглядит так:

**typedef** **struct** zveno \*svqz;

**typedef** **struct** zveno

{

**unsigned** **int** Element[256]; **// Множество из VS.**

svqz Sled; **// Указатель на узел.**

zveno(); **// Конструктор.**

};

zveno::zveno()

**//Обнуление элементов.**

{

**for**(**int** k=0;k<256;Element[k++]=0);

}

Тогда, например, фрагмент алгоритма:

**if** v и w принадлежат различным множествам W1 и W2 из VS

{

Заменить W1 и W2 на объединение W1 и W2 в VS;

Добавить (v,w) к T;

}

запишется на языке С++ так:

Res1 = Res2 = **NULL**;

Poisk (UkStr,T1,&Res1);

Poisk (UkStr,T2,&Res2);

**if** ( Res1!=Res2 )

{

**for** (**int** k=0;k<256;k++)

**if** ( Res1->Element[k]==1 || Res2->Element[k]==1 ) Res1->Element[k]=1;

Udalenie (&Res2,UkStr);

cout << "(" << T1 << " " << T2 << ") ";

}

Пример. ***Построение остовного дерева наименьшей стоимости (алгоритм Краскала).***

**#include** <iostream.h>

**#define** TRUE 1

**#define** FALSE 0

**typedef** **int** Boolean;

**typedef** **unsigned** **int** SubInt;

**typedef** **struct** Uzel \*Ref;

**typedef** **struct** Uzel

{

SubInt X; **//Начало дуги.**

SubInt Y; **//Конец дуги**

**int** Pay; **//Стоимость дуги.**

Ref Left; **//Указатель на левого сына.**

Ref Right;**//Указатель на правого сына.**

};

**typedef** **struct** zveno \*svqz;

**typedef** **struct** zveno

{

**unsigned** **int** Element[256];

svqz Sled;

zveno();

};

zveno::zveno()

{

**for**(**int** k=0;k<256;Element[k++]=0);

}

**class** Spisok

{

**private**:

Ref Root;

**void** Search (int, int, int, Ref \*);

**void** Poisk (svqz, SubInt, svqz \*);

**void** Postroenie (svqz \*);

**void** Udalenie (svqz \*, svqz);

**public**:

Spisok() { Root = **NULL**;} **//Вначале дерево пусто.**

**void** Reshenie();

**void** Postr();

};

**void** Spisok::Search (**int** A, **int** B, **int** C, Ref \*p)

**//Добавление вершины, содержащей поля A,B,C, в дерево \*p.**

{

**if** ( (\*p) == **NULL** )

{

(\*p) = **new** (Uzel); (\*\*p).X = A; (\*\*p).Y = B; (\*\*p).Pay = C;

(\*\*p).Left = (\*\*p).Right = **NULL**;

}

**else**

**if** ( C<=(\*\*p).Pay ) Search (A,B,C,&((\*\*p).Left));

**else**

**if** ( C>(\*\*p).Pay ) Search (A,B,C,&((\*\*p).Right));

}

**void** Spisok::Postroenie (svqz \*UkStr)

**//Постpоение линейного однонапpавленного списка**

**//с заглавным звеном, содержащего вершины графа.**

{

svqz UkZv;

**int** el;

(\*UkStr) = **new** (zveno);

UkZv = (\*UkStr); UkZv->Sled = **NULL**;

cout << "Вводите вершины графа: \n";

cin >> el;

**while** ( el!=0 )

{

UkZv->Sled = **new** (zveno); UkZv = UkZv->Sled;

UkZv->Element[el] = 1; UkZv->Sled = **NULL**;

cin >> el;

}

}

**void** Spisok::Postr()

**//Постpоение деpева, содержащего все ребра графа.**

{

**int** A,B,C;

cout << "Для каждого ребра вводите начальную, затем конечную\n";

cout << "вершины и стоимость ребра, их соединяющего:\n";

cin >> A >> B >> C;

**while** ( A!=0 )

{ Search (A,B,C,&Root);

cin >> A >> B >> C;

}

}

**void** Spisok::Poisk (svqz st, SubInt MENT, svqz \*Res)

{

svqz q;

(\*Res) = **NULL**; q = st->Sled;

**while** ( q != **NULL** && (\*Res) == **NULL** )

{

**if** ( q->Element[MENT]==1 ) (\*Res) = q;

**else** q = q->Sled;

}

}

**void** Spisok::Udalenie (svqz \*zv, svqz UkStr)

**//Удаление из однонапpавленного списка с заглавным звеном**

**//элемента, на который указывает указатель zv.**

{

svqz Z; **//"Стаpый" указатель.**

svqz UkZv1; **//"Hовый" указатель.**

**if** ( (\*zv)->Sled != **NULL** ) (\*\*zv) = \*((\*\*zv).Sled);

**else**

{ Z = UkStr; UkZv1 = UkStr->Sled;

**while** (UkZv1 != (\*zv))

{ Z = UkZv1; UkZv1 = UkZv1->Sled; }

Z->Sled = **NULL**; **delete** UkZv1;

}

}

**void** Spisok::Reshenie()

{

svqz UkStr; **//Указатель на список.**

Ref UkUzel; **//Рабочий указатель на узел дерева.**

Ref UkUzel1; **//Рабочий указатель на узел дерева.**

SubInt T1,T2;

svqz Res1,Res2;

**//Построение первоначальных множеств вершин графа.**

Postroenie (&UkStr);

cout <<"Ребра остовного дерева наименьшей стоимости: ";

**while** ( UkStr->Sled->Sled != **NULL** )

{

UkUzel1 = Root; **//"Отстающий" указатель.**

UkUzel = Root->Left; **//"Опережающий" указатель.**

**if** ( UkUzel== **NULL** )

{ **//Выбор в дереве ребра наименьшей стоимости и ...**

T1 = Root->X; T2 = Root->Y;

**//... удаление этого ребра из дерева.**

Root = Root->Right; **delete** UkUzel1;

}

**else**

{ **//Выбор в дереве ребра наименьшей стоимости и ...**

**while** ( UkUzel->Left != **NULL** )

{

UkUzel1 = UkUzel1->Left;

UkUzel = UkUzel->Left;

}

T1 = UkUzel->X; T2 = UkUzel->Y;

**//... удаление этого ребра из дерева.**

UkUzel1->Left = UkUzel->Right;

**delete** UkUzel;

}

**//Если v и w принадлежат различным**

**//множествам W1 и W2 из VS ...**

Res1 = Res2 = **NULL**;

Poisk (UkStr,T1,&Res1);

Poisk (UkStr,T2,&Res2);

**if** ( Res1!=Res2 )

{

**for** (**int** k=0;k<256;k++)

**if** ( Res1->Element[k]==1 || Res2->Element[k]==1 )

Res1->Element[k]=1;

Udalenie (&Res2,UkStr);

cout << "(" << T1 << " " << T2 << ") ";

}

}

}

**void** main ()

{

Spisok Tree;

Tree.Postr();

Tree.Reshenie();

}

Текст этой программы можно взять [здесь](http://khpi-iip.mipk.kharkiv.edu/library/datastr/book_sod/kgsu/din108_1.zip).

Тестовые примеры:

пример (A) заимствован из [2, с.200-201];

пример (B) заимствован из [3, с.198];

пример (C) заимствован из [4, с.341].

Для каждого ребра вводите начальную, затем конечную

вершины и стоимость ребра, их соединяющего:

(A) 1 7 1 (B) 1 2 9 (C) 1 3 2

3 4 3 2 3 10 2 4 4

2 7 4 2 7 8 2 5 8

3 7 9 1 3 6 1 2 1

2 3 15 1 7 1 3 4 9

4 7 16 1 6 4 3 6 7

4 5 17 6 4 5 8 10 5

1 2 20 7 4 7 4 8 6

1 6 23 3 4 2 5 8 4

5 7 25 4 5 11 4 7 3

5 6 28 7 6 3 1 4 6

6 7 36 0 0 0 6 7 7

0 0 0 6 9 2

7 9 8

7 10 9

9 10 3

0 0 0

Вводите вершины графа:

(A) 1 2 3 4 5 6 7 0

(B) 1 2 3 4 5 6 7 0

(С) 1 2 3 4 5 6 7 8 9 10 0

Ребра остовного дерева наименьшей стоимости:

(A) (1 7) (3 4) (2 7) (3 7) (4 5) (1 6)

(B) (1 7) (3 4) (7 6) (6 4) (2 7) (4 5)

(C) (1 2) (6 9) (1 3) (9 10) (4 7) (5 8) (2 4) (8 10) (4 8)

***Замечания.***

1. *В монографии [2, с.201-202] показано, что остовное дерево наименьшей стоимости для графа с* ***|E|*** *ребрами можно найти за время* ***O( |E|\*log |E|)*** *в общем случае и* ***O(|E|)****, если |E| достаточно велико по сравнению с числом узлов.*
2. *В некоторых ситуациях требуется построить остов не минимального, а максимального веса. К этой задаче также применим алгоритм Краскала. Следует только всюду минимальный вес заменить* ***максимальным****.*
3. *[5, с.424]. Напомним Вам, что под* ***стягиванием ребра*** *мы подразумеваем операцию удаления ребра* ***e*** *и отождествление его концевых вершин.*

***Алгоритм Прима*** *для определения минимального взвешенного остова в связном взвешенном графе* ***G=(V,E)****формулируется так: выбрать произвольную вершину* ***v*** *в графе* ***G****. Среди всех ребер, входящих в* ***v****, выбрать ребро* ***e1*** *с минимальным весом.* ***Стянем e1****, и пусть* ***G'*** *- полученный граф. Повторить эти действия для* ***G'*** *и продолжить до тех пор, пока не будет определено ребро* ***en-1****. Ребра* ***e1, e2, ..., en-1*** *образуют минимальный взвешенный остов графа* ***G****.*

1. *С задачей об остове минимального веса тесно связана* ***задача Штейнера*** *[4, с.62-63]. Мы отметим лишь, что* ***какие-либо эффективные алгоритмы, решающие задачу Штейнера неизвестны!***

(1) Уилсон Р. Введение в теорию графов. - М.: Мир, 1977. - 207 с.

(2) Ахо А., Хопкрофт Дж., Ульман Дж. Построение и анализ вычислительных алгоритмов. - М.: Мир, 1979. - 536 с.

(3) Липский В. Комбинаторика для программистов: Пер. с польск. - М.: Мир, 1988. - 213 с.

(4) Лекции по теории графов / Емеличев В.А., Мельников О.И., Сарванов В.И., Тышкевич Р.И. - М.: Наука, 1990. - 384 с.

(5) Свами М., Тхуласираман К. Графы, сети и алгоритмы. - М.: Мир, 1984. - 454 с.

На следующем шаге мы рассмотрим ***алгоритм построения фундаментального множества циклов***.

## Остовы. Построение фундаментального множества циклов

На этом шаге мы рассмотрим ***алгоритм построения фундаментального множества циклов***.

Тесно связана с задачей нахождения стягивающего дерева задача построения ***фундаментального множества циклов***.

Если к стягивающему дереву **(V,T)** графа **G=(V,E)** мы добавим произвольное ребро **e**, принадлежащее **E\T**, то нетрудно отметить, что возникший при этом подграф **(V, T *в объединении с* {e})** содержит в точности один цикл, который мы будем обозначать через **Ce**. Очевидно, что **Ce** содержит ребро **e**.

***Определение 1 [1, с.98].***

Множество **J={Ce: e *принадлежит* E\T}** будем называть ***фундаментальным множеством циклов*** графа **G**(относительно стягивающего дерева **(V,T)**).

Название "фундаментальный" связано с тем фактом, что каждый цикл графа **G** можно некоторым естественным способом получить из циклов множества **J**.

Введем для произвольных множеств **A** и **B** операцию **A+B = (*объединение* A *и* B)\(*пересечение* A *и* B)**.

***Определение 2.***

Множество **A+B** будем называть ***симметрической разностью*** множеств **A** и **B**.

***Определение 3 [1, с.98].***

Множество **C** ребер графа называется ***псевдоциклом***, если каждая вершина графа **(V,C)** имеет четную степень.

Примером псевдоцикла является пустое множество и произвольный цикл графа.

В монографии [1, c.99] доказана теорема о фундаментальном множестве циклов.

***Теорема.***

Пусть **G=(V,E)** - связный неориентированный граф, а **(V,T)** - его стягивающее дерево. Произвольный цикл графа**G** можно однозначно представить как симметрическую разность некоторого числа фундаментальных циклов. В общем случае произвольный псевдоцикл **C** графа **G** можно однозначно представить в виде симметрической разности вида:

http://khpi-iip.mipk.kharkiv.edu/library/datastr/book_sod/kgsu/ris109_2.jpg

Реализуем теперь алгоритм, описанный в [1, с.100-101]. Этот алгоритм основывается на поиске в глубину и имеет структуру, аналогичную рекурсивному алгоритму нахождения [стягивающего дерева](http://khpi-iip.mipk.kharkiv.edu/library/datastr/book_sod/kgsu/din_0107.html#1). Каждая новая вершина, встречающаяся в процессе поиска, помещается в стек, представленный массивом **Stack**, и удаляется из стека после использования. Очевидно, что стек всегда содержит последовательность вершин с рассматриваемой в данный момент вершины **v** до корня. Поэтому если анализируемое нами ребро **{v,u}** замыкает цикл (**WGN[v]>WGN[u]>0** и **u** не находится непосредственно под верхним элементом стека), то вершина **u** находится в стеке и цикл, замыкаемый ребром **{v,u}** представлен верхней группой элементов стека, начиная с **v** и кончая вершиной **u**.

Пример.

**#include** <iostream.h>

**#define** TRUE 1

**#define** FALSE 0

**typedef** **int** Boolean;

**typedef** **struct** Leader \*Lref; **// Тип: указатель на заголовочный узел.**

**typedef** **struct** Trailer \*Tref; **// Тип: указатель на дуговой узел.**

**//Описание типа заголовочного узла.**

**typedef** **struct** Leader

{

**int** Key; **//Имя заголовочного узла.**

**int** Count; **//Количество предшественников.**

**int** WGN; **//Характеристика узла.**

Tref Trail;**//Указатель на список смежности.**

Lref Next; **//Указатель на следующий узел в списке заголовочных узлов.**

};

**//Описание типа дугового узла.**

**typedef** **struct** Trailer

{

Lref Id;

Tref Next;

};

**class** Spisok

{

**private**:

Lref Head; **//Указатель на голову списка заголовочных узлов.**

Lref Tail; **//Указатель на фиктивный элемент**

**// в конце списка заголовочных узлов.**

Lref Stack[30]; **//Рабочий стек.**

**void** SearchGraph (int, Lref \*);

**void** Cycle (Lref, **int** \*, **int** \*);

**public**:

Spisok() {**//Инициализация списка заголовочных узлов.**

Head = Tail = **new** (Leader); }

**void** MakeGraph ();

**void** PrintGraph ();

**void** Mn\_Cycle();

};

**void** main ()

{

Spisok A;

**//Построение графа и вывод его структуры Вирта.**

A.MakeGraph ();

A.PrintGraph (); cout<<endl;

**//Построение множества фундаментальных циклов графа.**

A.Mn\_Cycle();

}

**void** Spisok::Mn\_Cycle()

**//Построение множества фундаментальных циклов графа G.**

{

cout << "Фундаментальные циклы:\n ";

**int** num = 0, d = 0; **//Количество элементов в стеке.**

Stack[0] = **NULL**;

Lref t = Head;

**while** ( t!=Tail )

{ t->WGN = 0; t = t->Next; }

t = Head;

**while** ( t!=Tail )

{

**if** ( t->WGN==0 ) Cycle(t,&d,&num);

t = t->Next;

}

}

**void** Spisok::Cycle (Lref r, **int** \*d, **int** \*num)

**//Нахождение фундаментального множества циклов для**

**//компоненты связности, содержащей вершину r->Key.**

{

Tref t;

**int** i;

(\*d)++; Stack[(\*d)] = r;

(\*num)++; r->WGN = \*num;

t = r->Trail;

**while** ( t != **NULL** )

{

**if** ( t->Id->WGN==0 ) Cycle (t->Id,d,num);

**else**

**if** (t->Id->Key != Stack[(\*d)-1]->Key &&

r->WGN > t->Id->WGN)

{

i = \*d;

**while** ( Stack[i]->Key != t->Id->Key )

{

cout << Stack[i]->Key << " ";

i--;

}

cout << t->Id->Key << endl;

}

t = t->Next;

}

**// Использованная вершина r->Key удаляется из стека.**

(\*d)--;

}

**void** Spisok::SearchGraph (**int** w, Lref \*h)

**//Функция возвращает указатель на заголовочный узел**

**//с ключом w в графе, заданном структурой Вирта с указателем Head.**

{

\*h = Head; (\*Tail).Key = w;

**while** ((\*\*h).Key!=w) \*h = (\*\*h).Next;

**if** (\*h==Tail)

**//В списке заголовочных узлов нет узла с ключом w.**

**//Поместим его в конец списка Head.**

{ Tail = **new** (Leader); (\*\*h).Count = 0;

(\*\*h).Trail = **NULL**; (\*\*h).Next = Tail; }

}

**void** Spisok::MakeGraph ()

**//Функция возвращает указатель Head на структуру**

**//Вирта, соответствующую ориентированному графу.**

{

**int** x,y;

Lref p,q; **//Рабочие указатели.**

Tref t,r; **//Рабочие указатели.**

Boolean Res; **//Флаг наличия дуги.**

cout<<"Вводите начальную вершину дуги: ";

cin>>x;

**while** (x!=0)

{

cout<<"Вводите конечную вершину дуги: "; cin>>y;

**//Определим, существует ли в графе дуга (x,y)?**

SearchGraph (x, &p); SearchGraph (y,&q);

r = (\*p).Trail; Res = FALSE;

**while** ((r!=**NULL**)&&(!Res))

**if** ((\*r).Id==q) Res = TRUE;

**else** r = (\*r).Next;

**if** (!Res) **//Если дуга отсутствует, то поместим её в граф.**

{ t = **new** (Trailer); (\*t).Id = q;

(\*t).Next = (\*p).Trail; (\*p).Trail = t; (\*q).Count++; }

cout<<"Вводите начальную вершину дуги: "; cin>>x;

}

}

**void** Spisok::PrintGraph ()

**//Вывод структуры Вирта, заданной указателем**

**//Head и соответствующей ориентированному графу.**

{

Lref p; **//Рабочий указатель.**

Tref q; **//Рабочий указатель.**

p = Head;

**while** (p!=Tail)

{

cout<<"("<<(\*p).Key; q = (\*p).Trail;

**while** (q!=**NULL**)

{ cout<<(\*(\*q).Id).Key; q = (\*q).Next; }

cout<<")"; p = (\*p).Next; cout<<" ";

}

}

Текст этой программы можно взять [здесь](http://khpi-iip.mipk.kharkiv.edu/library/datastr/book_sod/kgsu/din109_1.zip).

Результат работы приложения изображен на рисунке 1:

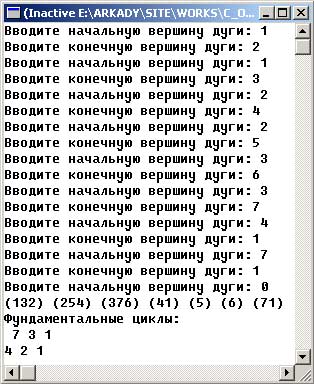


Рис.1. Результат работы приложения

Оценим теперь ***вычислительную сложность*** этого алгоритма [1, с.101].

Отметим сначала, что общее число шагов, как и во всех алгоритмах, основанных на поиске в глубину, имеет порядок **O(m+n)**. К этому следует прибавить суммарную длину всех циклов. Эта длина не превосходит **(m-n+1)n**, что дает общую сложность алгоритма **O(nm+n)**.

(1) Липский В. Комбинаторика для программистов: Пер. с польск. - М.: Мир, 1988. - 213 с.

Со следующего шага мы начнем знакомиться с ***алгоритмами с возвратом***.

## Алгоритмы с возвратом (общие сведения)

На этом шаге мы приведем ***общие сведения об алгоритмах с возвратом***.

До сих пор мы рассматривали только такие алгоритмы на графах, в процессе работы которых нам никогда не приходилось возвращаться ***назад***. Если, например, при обходе графа вершина была просмотрена, то она и оставалась в списке просмотренных вершин ***до окончания работы алгоритма***. Однако существует большой класс задач, для которых ***неизвестны такие алгоритмы***.

Пожалуй, наиболее известной такой задачей является ***задача коммивояжера*** (см., например, [1]) - задача о поиске кратчайшего маршрута между **N** городами, при этом лишь некоторые из городов непосредственно соединены дорогами.

Ниже мы рассмотрим задачу о поиске такого марщрута, который начинается и заканчивается в одном и том же городе и, кроме того, не заходит в один и тот же город дважды. Эта задача известна как ***задача поиска гамильтонова цикла в графе***.

В отличие от эйлеровых путей не известно ни одного простого необходимого и достаточного условия для существования гамильтоновых путей и это несмотря на то, что эта задача - одна из центральных в теории графов. Не известен также алгоритм, который проверял бы существование гамильтонова пути в произвольном графе, используя число шагов, ограниченное многочленом от переменной n (число вершин в графе). Имеются некоторые основания, но нет математического довазательства того, чтобы предполагать, что такое положение вещей вызвано не столько нашим незнанием, а скорее тем фактом, что такого алгоритма не существует.

Проблема существования гамильтонова пути принадлежит к классу так называемых **2NP-*полных задач*** [2, с.109]. Это широкий класс задач, включающий фундаментальные задачи из теории графов, логики, теории чисел, дискретной оптимизации и других дисциплин, ни для одной из которых неизвестен полиномиальный алгоритм (т.е. с числом шагов, ограниченным полиномом от размерности задачи), причем существование полиномиального алгоритма для хотя бы одной из них автоматически влекло бы за собой существование полиномиальных алгоритмов для всех этих задач. Именно факт фундаментальности многих **NP**-полных задач в различных областях и то, что, несмотря на независимые друг от друга усилия специалистов в этих областях, не удалось найти полиномиального алгоритма ни для одной из этих задач, склоняет к предположению, что такого алгоритма не существует.

(1) Рейнгольд Э., Hивергельт Ю., Део H. Комбинаторные алгоритмы. Теория и практика. - М.: Мир, 1980. - 476 с.

(2) Липский В. Комбинаторика для программистов: Пер. с польск. - М.: Мир, 1988. - 213 с.

На следующем шаге мы рассмотрим ***построение алгоритма с возвратом***.

## Построение алгоритмов с возвратом

На этом шаге мы рассмотрим ***построение алгоритма с возвратом***.

Опишем, следуя монографии [1], общий метод, позволяющий значительно сократить число шагов в алгоритмах типа полного перебора всех возможностей. Чтобы применить этот метод, искомое решение должно иметь вид последовательности **(x1, x2, ...,xn)**.

Основная идея метода состоит в том, что мы строим решение последовательно, начиная с пустой последовательности **e** (длины 0). Вообще, имея данное частичное решение **(x1, x2, ...,xi)**, мы стараемся найти такое допустимое значение **xi+1**, относительно которого мы не можем сразу заключить, что **(x1, x2, ...,xi+1)** можно расширить до некоторого решения (либо **(x1, x2, ...,xi+1)** уже является решением). Если такое предполагаемое, но еще не использованное решение **xi+1** существует, то мы добавляем его к нашему частичному решению и продолжаем процесс для последовательности **(x1, x2, ...,xi+1)**. Если его не существует, то мы возвращаемся к нашему частичному решению **(x1, x2, ...,xi-1)** и продолжаем наш процесс, отыскивая новое, еще не использованное допустимое значение **xi'** - отсюда название ***"алгоритм с возвратом"*** (англ. **Backtracking**) [1, с.110].

Точнее говоря, мы предполагаем, что для каждого **k>0** существует некоторое множество **Ak**, из которого мы будем выбирать кандидатов для **k**-й координаты частичного решения. Очевидно, что множества **Ak** должны быть определены таким образом, что для каждого **k<=n** множество **Ak** содержало элемент **xk** (на практике мы не можем вообще исключить ситуацию, когда множество **Ak** содержит некоторые "лишние" элементы, не появляющиеся в **k**-й координате ни одного целочисленного решения).

Мы предполагаем также, что существует некоторая простая функция, которая произвольному частичному решению **(x1, x2, ...,xi)** ставит в соответствие значение **P(x1, x2, ...,xi)** (***истина*** либо ***ложь***) таким образом, что если **P(x1, x2, ...,xi) = *ложь***, то последовательность **(x1, x2, ...,xi)** несомненно нельзя расширить до решения.

Если **P(x1, x2, ...,xi) = *истина***, то мы говорим, что значение **xi** ***допустимо*** (для частичного решения **(x1, x2, ...,xi-1)**, но это отнюдь не означает, что **(x1, x2, ...,xi-1)** обязательно расширяется до полного решения. Этот процесс можно записать в виде следующей схемы [1, с.111]:

{

k = 1;

**while** ( k>0 )

**if** ( существует еще неиспользованный элемент yпринадлежащий Ak, такой что

P(X[1],X[2],...,X[k-1],y)

{

X[k] = y; **//Элемент y использован.**

**if** ((X[1],X[2],...,X[k]) является целочисленным решением)

{

cout << (X[1],X[2],...,X[k]);

k++;

}

}

**else**

**//Возврат на более короткое частичное решение; все эле-**

**//менты множества Ak вновь становятся неиспользованными.**

k -= 1;

}

Этот алгоритм находит все решения в предположении, что множества **Ak** конечные и что существует **n** такое, что **P(x1, x2, ...,xn) = *ложь*** для всех **x1**, принадлежащем **A1**, **x2**, принадлежащем **A2**, ...,**xn**, принадлежащем **An** (последнее условие означает, что все решения имеют длину меньше **n**).

Приведем рекурсивный вариант схемы алгоритма с возвратом [1, с.112]:

AP (k);

**//Генерирование всех решений, являющихся расширением**

**//последовательности X[1],X[2],...,X[k-1].**

**//Массив X - глобальный.**

{

**for** (y, принадлежащего Ak, такого, что P(X[1],X[2],...,X[k-1]))

{

X[k] = y;

**if** (X[1],X[2],...,X[k] есть целочисленное решение)

{

cout << X[1],X[2],...,X[k];

AP(k+1);

}

}

Генерирование всех целочисленных решений можно вызвать вызовом AP(1). Представление алгоритма с возвратом мы начали с несколько более сложного нерекурсивного варианта только потому, что в рекурсивном варианте "возврат" не появляется в явном виде, будучи частью реализации механизма рекурсии.

Пример [1,c.118]. Для данных целых чисел **a1, a2, ...,an** нахождение множества индексов **J**, принадлежащего **{1, 2, ..., n}**, такого, что сумма **aj** равна **b** при **j**, принадлежащих **J**, если такое множество существует.

**#include** <iostream.h>

**#define** TRUE 1

**#define** FALSE 0

**#define** Node 20 **//Максимальное количество вершин в графе.**

**typedef** **int** Boolean;

**typedef** **struct** L \*Lref; **// Тип: указатель на заголовочный узел.**

**typedef** **struct** T \*Tref; **// Тип: указатель на дуговой узел.**

**//Описание типа заголовочного узла.**

**typedef** **struct** L

{

**int** Key; **//Имя заголовочного узла.**

**int** Count; **//Количество предшественников.**

Boolean Flag; **//Флаг посещения узла при обходе.**

Tref Trail; **//Указатель на список смежности.**

Lref Next; **//Указатель на следующий узел в списке заголовочных узлов.**

};

**//Описание типа дугового узла.**

**typedef** **struct** T

{

Lref Id;

Tref Next;

};

**class** Spisok

{

**private**:

Lref Head; **//Указатель на голову списка заголовочных узлов.**

Lref Tail; **//Указатель на фиктивный элемент**

**// в конце списка заголовочных узлов.**

**int** X[Node+1];

**void** SearchGraph (int, Lref \*);

**public**:

Spisok() {**//Инициализация списка заголовочных узлов.**

Head = Tail = **new** (L); }

Lref GetHead() { **return** Head; }

Lref GetTail() { **return** Tail; }

**void** MakeGraph ();

**void** PrintGraph ();

**void** Summa (int, int, int);

**void** X1 (Lref t) {X[1] = t->Key;};

};

**void** main ()

{

Spisok A;

Lref t; **//Рабочий указатель для перемещения**

**// по списку заголовочных звеньев.**

Lref t1;

**int** n=0,B;

**//Построение графа и вывод его структуры Вирта.**

A.MakeGraph ();

A.PrintGraph (); cout<<endl;

**//Подсчет n - количества вершин в графе Head.**

t = A.GetHead();

**while** (t!=A.GetTail())

{ n++; t = (\*t).Next; }

**// ------------------------------------**

**//Обнаружение множества индексов.**

cout << "Введите число B: "; cin >> B;

t = A.GetHead();

**while** (t!=A.GetTail())

{

**//Инициализация.**

t1 = A.GetHead();

**while** (t1!=A.GetTail())

{ t1->Flag = TRUE; t1 = t1->Next; }

A.X1(t); t->Flag = FALSE;

A.Summa (2,B,n);

t = t->Next;

}

}

**void** Spisok::SearchGraph (**int** w, Lref \*h)

**//Функция возвращает указатель на заголовочный узел**

**//с ключом w в графе, заданном структурой Вирта с указателем Head.**

{

\*h = Head; (\*Tail).Key = w;

**while** ((\*\*h).Key!=w) \*h = (\*\*h).Next;

**if** (\*h==Tail)

**//В списке заголовочных узлов нет узла с ключом w.**

**//Поместим его в конец списка Head.**

{ Tail = **new** (L); (\*\*h).Count = 0;

(\*\*h).Trail = **NULL**; (\*\*h).Next = Tail; }

}

**void** Spisok::MakeGraph ()

**//Функция возвращает указатель Head на структуру**

**//Вирта, соответствующую ориентированному графу.**

{

**int** x,y;

Lref p,q; **//Рабочие указатели.**

Tref t,r; **//Рабочие указатели.**

Boolean Res; **//Флаг наличия дуги.**

cout<<"Вводите начальную вершину дуги: ";

cin>>x;

**while** (x!=0)

{

cout<<"Вводите конечную вершину дуги: "; cin>>y;

**//Определим, существует ли в графе дуга (x,y)?**

SearchGraph (x, &p); SearchGraph (y,&q);

r = (\*p).Trail; Res = FALSE;

**while** ((r!=**NULL**)&&(!Res))

**if** ((\*r).Id==q) Res = TRUE;

**else** r = (\*r).Next;

**if** (!Res) **//Если дуга отсутствует, то поместим её в граф.**

{ t = **new** (T); (\*t).Id = q;

(\*t).Next = (\*p).Trail; (\*p).Trail = t; (\*q).Count++; }

cout<<"Вводите начальную вершину дуги: "; cin>>x;

}

}

**void** Spisok::PrintGraph ()

**//Вывод структуры Вирта, заданной указателем**

**//Head и соответствующей ориентированному графу.**

{

Lref p; **//Рабочий указатель.**

Tref q; **//Рабочий указатель.**

p = Head;

**while** (p!=Tail)

{

cout<< (\*p).Key << "("; q = (\*p).Trail;

**while** (q!=**NULL**)

{ cout<<(\*(\*q).Id).Key; q = (\*q).Next; }

cout<<")"; p = (\*p).Next; cout<<" ";

}

}

**void** Spisok::Summa (**int** k, **int** B, **int** n)

**//Нахождение множества вершин в графе, заданном**

**//структурой Вирта с указателем Head.**

{

**int** i; **//Параметр цикла.**

Lref t; **//Указатель на k-ю вершину частичного решения.**

Tref r;

**int** S;

**int** Set1[256]; **//Вспомогательное множество.**

**int** m; **//Количество элементов в Set1.**

SearchGraph (X[k-1], &t);

r = t->Trail;

**while** ( r != **NULL** )

{

X[k] = r->Id->Key;

**//Построение вспомогательного множества Set1 для**

**//проверки, все ли элементы массива X различны.**

**for** (**int** j=0;j<256;Set1[j++]=0);

**for** (i=1;i<=k;i++) Set1[X[i]]=1;

**//Подсчет m - количества элементов в множестве Set1.**

m = 0;

**for**(i=1;i<=Node;i++)

**if** (Set1[i]==1) m++;

**// -------------------------------**

S = 0;

**for** (i=1;i<=k;i++) S += X[i];

**// ----------------------------**

**if** (k<n+1 && S==B && m==k)

{

**//Вывод множества индексов на экран дисплея.**

**for** (i=1;i<=k;i++) cout << X[i] << " ";

cout << endl;

}

**else**

**if** (r->Id->Flag)

{

X[k] = r->Id->Key;

r->Id->Flag = FALSE;

Summa (k+1,B,n);

r->Id->Flag = TRUE;

}

r = r->Next;

}

}

Текст этой программы можно взять [здесь](http://khpi-iip.mipk.kharkiv.edu/library/datastr/book_sod/kgsu/din111_1.zip).

(1) Липский В. Комбинаторика для программистов: Пер. с польск. - М.: Мир, 1988. - 213 с.

На следующем шаге мы познакомимся с ***гамильтоновыми циклами***.

## Гамильтоновы циклы

На этом шаге мы рассмотрим ***гамильтоновы циклы***.

Отметим, что начало исследований так называемых ***гамильтоновых графов*** относится к графам многогранников [1, с.70].

В 1857 г. ирландский математик ***Гамильтон*** предложил игру, названную "путешествие по додекаэдру". Игра сводилась к обходу по ребрам всех вершин правильного додекаэдра при условии, что ни в одну из вершин нельзя заходить более одного раза. ***Додекаэдр*** - это многогранник, гранями которого служат 12 правильных пятиугольников. У него 20 вершин и 30 ребер. Вершины и ребра додекаэдра составляют некоторый плоский граф.

***Определение.***

***Простой цикл*** в графе - это замкнутый путь, все вершины которого, кроме **v0** и **vn**, попарно различны.

***Гамильтонов цикл*** - это простой цикл, содержащий все вершины графа. Заметим, что гамильтонов цикл есть далеко не в каждом графе. Граф называется ***гамильтоновым***, если в нем имеется гамильтонов цикл.

Граф, который содержит простой путь, проходящий через каждую его вершину, называется ***полугамильтоновым*** [2, с.48-49]. Ясно, что всякий гамильтонов граф является полугамильтоновым.

Применим теперь алгоритм с возвратом для нахождения гамильтонова цикла в графе.

Алгоритм, с помощью которого мы будем искать гамильтоновы циклы, называется ***поиск с возвращением***. В основе его лежит понятие ***частичного решения***. Пусть решение задачи представляется в виде некоторой последовательности. В нашем случае это просто последовательность всех вершин графа, удовлетворяющая очевидным ограничениям. Начальный отрезок последовательности, который удовлетворяет ограничениям, определяющим полное решение, называется ***частичным решением***.

***Частичным решением для гамильтонова цикла*** является любая последовательность вершин, определяющая простой путь.

Если в последовательность, представляющую частичное решение, добавить новый элемент так, что расширенная последовательность снова будет частичным решением, то новая последовательность называется ***продолжением частичного решения***. Продолжение, как правило, неоднозначно.

Идея поиска с возвращением состоит в том, чтобы, начав с тривиального частичного решения, последовательно продолжать его до тех пор, пока не будет получено полное решение, либо продолжение станет невозможным. В последнем случае мы возвращаемся на шаг назад и пробуем построить другое продолжение. Если пространство поиска конечно, то либо мы получим полное решение, либо убедимся в невозможности его построения, поскольку осуществлен полный перебор возможных продолжений.

После этих замечаний приведем ***реализацию алгоритма поиска гамильтонова пути в связном неориентированном графе***:

Программа 1.

**//Нахождение всех гамильтоновых циклов в графе,**

**//заданном структурой Вирта.**

**#include** <iostream.h>

**#define** TRUE 1

**#define** FALSE 0

**#define** Node 20 **//Максимальное количество вершин в графе.**

**typedef** **int** Boolean;

**typedef** **struct** L \*Lref; **// Тип: указатель на заголовочный узел.**

**typedef** **struct** T \*Tref; **// Тип: указатель на дуговой узел.**

**//Описание типа заголовочного узла.**

**typedef** **struct** L

{

**int** Key; **//Имя заголовочного узла.**

**int** Count; **//Количество предшественников.**

Boolean Flag; **//Флаг посещения узла при обходе.**

Tref Trail; **//Указатель на список смежности.**

Lref Next; **//Указатель на следующий узел в списке заголовочных узлов.**

};

**//Описание типа дугового узла.**

**typedef** **struct** T

{

Lref Id;

Tref Next;

};

**//Описание типа узла стека.**

**typedef** Lref TipElement;

**typedef** **struct** Zveno \*svqz;

**typedef** **struct** Zveno

{

TipElement Element; **//Указатель на список смежности.**

svqz Sled;

};

**class** Spisok

{

**private**:

Lref Head; **//Указатель на голову списка заголовочных узлов.**

Lref Tail; **//Указатель на фиктивный элемент**

**// в конце списка заголовочных узлов.**

**int** X[Node+1]; **//Результат работы программы.**

**void** SearchGraph (int, Lref \*);

**public**:

Spisok() {**//Инициализация списка заголовочных узлов.**

Head = Tail = **new** (L); }

Lref GetHead() { **return** Head; }

Lref GetTail() { **return** Tail; }

**void** MakeGraph ();

**void** PrintGraph ();

**void** Hamilton (int, int);

**void** X1 (Lref t) {X[1] = t->Key;};

};

**void** main ()

{

Spisok A;

Lref t; **//Рабочий указатель для перемещения**

**// по списку заголовочных звеньев.**

**int** n=0;

**//Построение графа и вывод его структуры Вирта.**

A.MakeGraph ();

A.PrintGraph (); cout<<endl;

**//Инициализация и подсчет количества вершин графа.**

t = A.GetHead();

**while** (t!=A.GetTail())

{ t->Flag = TRUE; n++; t = (\*t).Next; }

**// ------------------------------------**

**//Нахождение всех гамильтоновых циклов.**

t = A.GetHead();

A.X1(t);

t->Flag = FALSE;

A.Hamilton (2,n);

}

**void** Spisok::SearchGraph (**int** w, Lref \*h)

**//Функция возвращает указатель на заголовочный узел**

**//с ключом w в графе, заданном структурой Вирта с указателем Head.**

{

\*h = Head; (\*Tail).Key = w;

**while** ((\*\*h).Key!=w) \*h = (\*\*h).Next;

**if** (\*h==Tail)

**//В списке заголовочных узлов нет узла с ключом w.**

**//Поместим его в конец списка Head.**

{ Tail = **new** (L); (\*\*h).Count = 0;

(\*\*h).Trail = **NULL**; (\*\*h).Next = Tail; }

}

**void** Spisok::MakeGraph ()

**//Функция возвращает указатель Head на структуру**

**//Вирта, соответствующую ориентированному графу.**

{

**int** x,y;

Lref p,q; **//Рабочие указатели.**

Tref t,r; **//Рабочие указатели.**

Boolean Res; **//Флаг наличия дуги.**

cout<<"Вводите начальную вершину дуги: ";

cin>>x;

**while** (x!=0)

{

cout<<"Вводите конечную вершину дуги: "; cin>>y;

**//Определим, существует ли в графе дуга (x,y)?**

SearchGraph (x, &p); SearchGraph (y,&q);

r = (\*p).Trail; Res = FALSE;

**while** ((r!=**NULL**)&&(!Res))

**if** ((\*r).Id==q) Res = TRUE;

**else** r = (\*r).Next;

**if** (!Res) **//Если дуга отсутствует, то поместим её в граф.**

{ t = **new** (T); (\*t).Id = q;

(\*t).Next = (\*p).Trail; (\*p).Trail = t; (\*q).Count++; }

cout<<"Вводите начальную вершину дуги: "; cin>>x;

}

}

**void** Spisok::PrintGraph ()

**//Вывод структуры Вирта, заданной указателем**

**//Head и соответствующей ориентированному графу.**

{

Lref p; **//Рабочий указатель.**

Tref q; **//Рабочий указатель.**

p = Head;

**while** (p!=Tail)

{

cout<< (\*p).Key << "("; q = (\*p).Trail;

**while** (q!=**NULL**)

{ cout<<(\*(\*q).Id).Key; q = (\*q).Next; }

cout<<")"; p = (\*p).Next; cout<<" ";

}

}

**void** Spisok::Hamilton (**int** k, **int** n)

**//Нахождение всех гамильтоновых циклов в графе, структура**

**//смежности которого заданна указателем t.**

{

**int** i; **//Параметр цикла.**

Lref t; **//Указатель на k-ю вершину частичного решения.**

Tref r;

SearchGraph (X[k-1], &t);

r = t->Trail;

**while** ( r != **NULL** )

{

X[k] = r->Id->Key;

**if** (k==n+1 && X[k]==X[1])

{

**for** (i=1;i<=n;i++) cout << X[i] << " ";

cout << X[1] << endl;

}

**else**

**if** (r->Id->Flag)

{

r->Id->Flag = FALSE;

Hamilton (k+1,n);

r->Id->Flag = TRUE;

}

r = r->Next;

}

}

Текст этой программы можно взять [здесь](http://khpi-iip.mipk.kharkiv.edu/library/datastr/book_sod/kgsu/din112_1.zip).

Результат работы приложения изображен на рисунке 1:

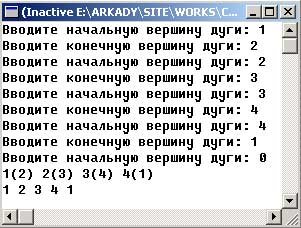


Рис.1. Результат работы приложения

Программа [3, с.134-135].

(DEFUN HAMILTCYCLE (LAMBDA (GRAPH)

**; Построение гамильтонова цикла:**

**; GRAPH - граф, представленный структурой смежности в**

**; виде ассоциативного списка,**

**; Результат: гамильтонов цикл в виде списка вершин,**

**; NIL, если гамильтонова цикла не существует**

(COND ( (**NULL** GRAPH) NIL )

( T (COND ( (**NULL** (CDR GRAPH))

(LIST (CAAR GRAPH)))

( T (HC GRAPH (CAAR GRAPH)

(LIST (CAAR GRAPH))

(CDAR GRAPH)) )) ))

))

**; ------------------------------------------**

(DEFUN HC (LAMBDA (GRAPH START VISITED SONS)

**; GRAPH - граф, представленный структурой смежности в**

**; виде ассоциативного списка,**

**; START - вершина графа - начало гамильтонова цикла,**

**; VISITED - список посещенных вершин графа,**

**; SONS - список вершин, смежных первой вершине в списке VISITED**

(COND ( (**NULL** SONS) NIL )

( T (COND ( (AND (MEMBER START SONS)

(EQ (LENGTH GRAPH)

(LENGTH VISITED)))

(REVERSE VISITED) )

( T (COND

( (MEMBER (CAR SONS) VISITED)

(HC GRAPH START VISITED

(CDR SONS)) )

( T (OR (HC GRAPH START

(CONS

(CAR SONS) VISITED)

(NEIGHBOUR3 (CAR SONS) GRAPH)

)

(HC

GRAPH START VISITED

(CDR SONS))) )) )) ))

))

**; ---------------------------------**

(DEFUN NEIGHBOUR3 (LAMBDA (X GRAPH)

**; Функция возвращает список вершин графа GRAPH,**

**; смежных с вершиной X**

(COND ( (**NULL** (ASSOC X GRAPH)) NIL )

( T (CDR (ASSOC X GRAPH)) ))

))

Текст этой библиотеки можно взять [здесь](http://khpi-iip.mipk.kharkiv.edu/library/datastr/book_sod/kgsu/din112_2.zip).

Тестовые примеры:

$ (HAMILTCYCLE '((1 . (2 6)) (2 . (1 3 4)) (3 . (2 4))

(4 . (2 3 5)) (5 . (4 6)) (6 . (1 5))))

(1 2 3 4 5 6)

$ (HAMILTCYCLE '((A . (F D C)) (F . (E D A B))

(E . (D F B)) (D . (E F A)) (C . (A B)) (B . (F E C))))

(A F D E B C)

Разберем работу функции **HAMILTCYCLE**.

Для пустого графа она возвращает пустой список, а для графа, состоящего из единственной вершины - список, содержащий эту вершину.

Если граф состоит более чем из двух вершин, то вызывается функция **HC**. Именно эта функция осуществляет продолжение ***частичного решения***.

Частичное решение представлено в виде списка **VISITED** - это список просмотренных вершин в порядке, обратном порядку их просмотра.

Список **SONS** - это список соседей первой вершины в списке **VISITED**. С точки зрения частичного решения эта вершина является как раз последней. Поэтому вершины из списка **SONS** могут входить в продолжения частичного решения.

Если этот список пуст, то продолжение частичного решения невозможно и функция **HC** возвращает значение **NIL**.

В противном случае проверяется, нельзя ли на следующем шаге замкнуть гамильтонов цикл. Это возможно, если,

* ***во-первых***, начальная вершина содержится в списке **SONS** и,
* ***во-вторых***, если пройдены все вершины графа. Поскольку каждая вершина по условию посещается не более одного раза (это следует из определения частичного решения), то достаточно проверить, что длины списков **GRAPH** и **VISITED** равны.

Если оба вышеупомянутых условия выполнены, то список **VISITED** представляет собой искомый цикл.

Перейдем теперь к случаю, когда продолжение возможно, но не приводит немедленно к построению полного решения. Наша цель продолжить частичное решение. Для этого мы просматриваем список **SONS** до тех пор, пока не встретим вершину, которой нет в списке **VISITED**. Если такая вершина не найдена, то продолжение невозможно.

В противном случае вычисляется выражение

(OR (HC GRAPH START (CONS (CAR SONS) VISITED)

(NEIGHBOUR3 (CAR SONS) GRAPH))

(HC GRAPH START VISITED (CDR SONS)))

Это ключевой момент поиска с возвращением. Если при первом вызове функции **HC** вычисляется значение, отличное от **NIL**, то это означает, что продолжение **(CONS (CAR SONS) VISITED)** достроено до полного решения. Это решение и возвращается на верхний уровень. В противном случае с помощью второго вызова функции **HC**делается попытка найти продолжение, отличное от **(CONS (CAR SONS) VISITED)**.

Обратите внимание на то, что весь механизм возврата оказался "спрятанным" в рекурсию. Это позволило сделать программу ясной и компактной.

Функция **HAMILTCYCLE** прекратит работу, как только найдет первый гамильтонов цикл или просмотрит все возможности.

Заметим, что алгоритм поиска с возвращением можно интерпретировать как поиск в графе. Вершинами этого графа являются частичные решения, а ребрами - те пары частичных решений, одно из которых является непосредственным продолжением другого. Выбор того или иного алгоритма поиска на графе определяет стратегию поиска с возвращением.

***Замечания.***

1. *Следующие четыре неориентированных графа демонстрируют отсутствие* ***тесной*** *взаимосвязи между существованием эйлеровых и гамильтоновых циклов:*
2. 1) эйлеров и гамильтонов:
3. {1,2},{2,3},{3,4},{4,1};
4. 2) неэйлеровы и гамильтоновы:
5. a) {1,2},{2,3},{3,4},{2,4},{4,5},{5,1};
6. b) {1,2},{2,3},{3,4},{2,4},{4,6},{1,6},{4,5},{5,6};
7. 3) эйлеровы и негамильтоновы:
8. a) {1,2},{2,3},{3,4},{4,1},{4,5},{5,2},{2,6},{6,4};
9. b) {1,2},{2,3},{3,5},{5,6},{6,4},{1,4},{2,7},{7,4},{2,8},
10. {8,5},{4,5};
11. 4) неэйлеров и негамильтонов:
12. {1,2},{2,3},{3,4},{4,1},{4,5},{5,2}.

*Однако, двойственность между эйлеровыми и гамильтоновыми циклами (замена вершины на ребро и наоборот) приводит к тесной связи между этими двумя понятиями в применении к неориентированному графу* ***G*** *и соответствующему ему реберному графу, определяемому ниже.*

***Определение [4, с.237]****.*

*Реберный граф* ***Gl*** *графа* ***G*** *имеет столько же вершин, сколько ребер у графа* ***G****. Ребро между двумя вершинами графа* ***Gl*** *существует тогда и только тогда, когда ребра графа* ***G****, соответствующие этим двум вершинам, смежны (т.е. инцидентны одной и той же вершине графа* ***G****).*

*Верны два следующих утверждения о взаимоотношении между эйлеровыми и гамильтоновыми циклами, принадлежащие* ***Ф.Харари*** *[4, с.237].*

*1. Если граф* ***G*** *имеет эйлеров цикл, то граф* ***Gl*** *имеет как эйлеров, так и гамильтонов циклы.*

*2. Если граф* ***G*** *имеет гамильтонов цикл, то граф* ***Gl*** *также имеет гамильтонов цикл.*

*Обращение этих утверждений неверно!*

1. *Пусть* ***G*** *- граф с* ***n*** *вершинами* ***v1, v2, ...,vn****, степени* ***di*** *которых удовлетворяют неравенствам* ***d1<=d2<= ... <=dn-1<=dn****.*

*Граф* ***G*** *имеет* ***гамильтонов цикл****, если выполняется одно из следующих условий:*

* + ***условие Дирака****:* ***d1>=n/2****;*
  + ***условие Поша****:* ***dk >= k+1*** *для* ***k<n/2****;*
  + ***условие Бонди****: из* ***dl<=l*** *и* ***dk<=k*** *следует, что* ***dl+dk>=n (k<>l)****;*
  + ***условие Хватала****: из* ***dk<=k<=n/2*** *следует, что* ***dn-k>=n-k****.*

*Обратим также Ваше внимание на условие Дирака в другой формулировке.*

***Теорема Дирака****.*

*Обозначим* ***R(v)*** *- степень вершины* ***v*** *в графе. Если в простом графе с n>=3 вершинами* ***R(v)>=n/2*** *для любой вершины* ***v****, то граф является гамильтоновым.*

1. *Каждый гамильтонов цикл можно характеризовать с помощью* ***матрицы смежностей******(xij)*** *размера* ***n\*n****, имея в виду, что* ***xij=1****, если ребро* ***(i,j)*** *принадлежит* ***C****, и* ***xij=0*** *в противном случае.*
2. ***Толщиной*** *графа* ***G*** *(обозначается* ***l(G)****) называется число вершин в самой длинной простой цепи графа. Если граф* ***G*** *гамильтонов, то толщина* ***l(G)*** *равна числу вершин графа.*
3. *[5, с.48]. Простейший пример негамильтонова графа - это* ***тэта-граф****, графическое изображение которого похоже на греческую букву "тэта":*
4. o --------- o
5. ¦ ¦
6. o --- o --- o
7. ¦ ¦
8. o --------- o

(1) Емеличев В.А., Ковалев М.М., Кравцов М.К. Многогранники, графы, опитимизация. - М.: Наука, 1981. - 341 с.

(2) Уилсон И.Р., Эддиман А.М. Практическое введение в ПАСКАЛЬ. - М.: Радио и связь, 1983. - 143 с.

(3) Крюков А.П., Радионов А.Я., Таранов А.Ю., Шаблыгин Е.М. Программирование на языке R-Лисп. - М.: Радио и связь, 1991. - 192 с.

(4) Кристофидес Н. Теория графов. Алгоритмический подход. - М.: Мир, 1978. - 432 с.

(5) Березина Л.Ю. Графы и их применение. - М.: Просвещение, 1979. - 143 с.

На следующем шаге мы познакомимся с ***кликами***.

## Клики

На этом шаге мы рассмотрим ***понятие и реализацию кликов***.

Понятие ***клики*** используется в различных социологических теориях (вопросы, связанные с голосованием, альянсами и т.п.), а также в теории игр.

Вначале напомним Вам, что:

* ***часть графа*** **G=(V,E)** - это такой граф **G'=(V',E')**, что **V'** принадлежит **V** и **E'** принадлежит **E**;
* ***подграфом графа* G** называется такая его часть **G'**, которая вместе со всякой парой вершин **u,v** содержит и ребро**{u,v}**, если только оно есть в **G**;
* граф называется ***полным***, если любые две его различные вершины соединены ребром.

***Определение [1, с.34]***

***Клика графа*** - это любой максимальный полный подграф.

Не требует никакой изобретательности следующий процесс нахождения клики.

Пусть вершины графа **L=(X,U)** пронумерованы: **X={x1, x2, ..., xn}**. Выберем вершину **x1**, затем первую смежную с ней **xi**, потом вторую **xj**, смежную с обеими **(1<i<j)** и т.д. пока возможно, и пусть **p** - число вершин выявленной таким образом максимальной (по включению) клики. Попытаемся увеличить это число, рассматривая другой вариант процесса: вместо вершины **xk**, добавленной к некоторой клике на последнем шаге, добавляем другую вершину **xl**, **l>k**, тоже смежную со всеми вершинами этой клики (если такая **xl** есть), в надежде, что продолжение процесса по новой ветви окажется более успешным и даст хотя бы **(p+1)**-клику; если ни одна замена вершины **xk** не приводит к цели, возвращаемся еще на один шаг и т.д.

Пример.

**//Нахождение всех клик в графе, заданном структурой Вирта.**

**#include** <iostream.h>

**#define** TRUE 1

**#define** FALSE 0

**#define** Node 20 **//Максимальное количество вершин в графе.**

**typedef** **int** Boolean;

**typedef** **struct** L \*Lref; **// Тип: указатель на заголовочный узел.**

**typedef** **struct** T \*Tref; **// Тип: указатель на дуговой узел.**

**//Описание типа заголовочного узла.**

**typedef** **struct** L

{

**int** Key; **//Имя заголовочного узла.**

**int** Count; **//Количество предшественников.**

Boolean Flag; **//Флаг посещения узла при обходе.**

Tref Trail; **//Указатель на список смежности.**

Lref Next; **//Указатель на следующий узел в списке заголовочных узлов.**

};

**//Описание типа дугового узла.**

**typedef** **struct** T

{

Lref Id;

Tref Next;

};

**class** Spisok

{

**private**:

Lref Head; **//Указатель на голову списка заголовочных узлов.**

Lref Tail; **//Указатель на фиктивный элемент**

**// в конце списка заголовочных узлов.**

**int** X[20]; **//Результат работы программы.**

**void** SearchGraph (int, Lref \*);

**public**:

Spisok() {**//Инициализация списка заголовочных узлов.**

Head = Tail = **new** (L); }

Lref GetHead() { **return** Head; }

Lref GetTail() { **return** Tail; }

**void** MakeGraph ();

**void** PrintGraph ();

**void** Clique (int, int);

**void** X1 (Lref t) {X[1] = t->Key;};

};

**void** main ()

{

Spisok A;

Lref t; **//Рабочий указатель для перемещения**

**// по списку заголовочных звеньев.**

Lref t1;

**int** n=0;

**//Построение графа и вывод его структуры Вирта.**

A.MakeGraph ();

A.PrintGraph (); cout<<endl;

**//Инициализация и подсчет количества вершин графа.**

t = A.GetHead();

**while** (t!=A.GetTail())

{ n++; t = (\*t).Next; }

**// ------------------------------------**

**//Нахождение всех клик в графе Head.**

**for** (**int** i=3;i<=n;i++)

{

**//i - количество вершин в клике (i<=3).**

cout << "Перечислим все клики длины " << i << endl;

t = A.GetHead();

**while** (t!=A.GetTail())

{

**//Инициализация.**

t1 = A.GetHead();

**while** (t1!=A.GetTail())

{ t1->Flag = TRUE; t1 = t1->Next; }

A.X1(t); t->Flag = FALSE;

**//Отыскание клики с i вершинами, "начинающейся" в вершине t.**

A.Clique (2,i); t = t->Next;

}

}

}

**void** Spisok::SearchGraph (**int** w, Lref \*h)

**//Функция возвращает указатель на заголовочный узел**

**//с ключом w в графе, заданном структурой Вирта с указателем Head.**

{

\*h = Head; (\*Tail).Key = w;

**while** ((\*\*h).Key!=w) \*h = (\*\*h).Next;

**if** (\*h==Tail)

**//В списке заголовочных узлов нет узла с ключом w.**

**//Поместим его в конец списка Head.**

{ Tail = **new** (L); (\*\*h).Count = 0;

(\*\*h).Trail = **NULL**; (\*\*h).Next = Tail; }

}

**void** Spisok::MakeGraph ()

**//Функция возвращает указатель Head на структуру**

**//Вирта, соответствующую ориентированному графу.**

{

**int** x,y;

Lref p,q; **//Рабочие указатели.**

Tref t,r; **//Рабочие указатели.**

Boolean Res; **//Флаг наличия дуги.**

cout<<"Вводите начальную вершину дуги: ";

cin>>x;

**while** (x!=0)

{

cout<<"Вводите конечную вершину дуги: "; cin>>y;

**//Определим, существует ли в графе дуга (x,y)?**

SearchGraph (x, &p); SearchGraph (y,&q);

r = (\*p).Trail; Res = FALSE;

**while** ((r!=**NULL**)&&(!Res))

**if** ((\*r).Id==q) Res = TRUE;

**else** r = (\*r).Next;

**if** (!Res) **//Если дуга отсутствует, то поместим её в граф.**

{ t = **new** (T); (\*t).Id = q;

(\*t).Next = (\*p).Trail; (\*p).Trail = t; (\*q).Count++; }

cout<<"Вводите начальную вершину дуги: "; cin>>x;

}

}

**void** Spisok::PrintGraph ()

**//Вывод структуры Вирта, заданной указателем**

**//Head и соответствующей ориентированному графу.**

{

Lref p; **//Рабочий указатель.**

Tref q; **//Рабочий указатель.**

p = Head;

**while** (p!=Tail)

{

cout<< (\*p).Key << "("; q = (\*p).Trail;

**while** (q!=**NULL**)

{ cout<<(\*(\*q).Id).Key; q = (\*q).Next; }

cout<<")"; p = (\*p).Next; cout<<" ";

}

}

**void** Spisok::Clique (**int** k, **int** m)

**//Нахождение всех клик, содержащих m вершин, в графе,**

**//заданном структурой Вирта с указателем Head,**

**//k - количество вершин в частичном решении.**

{

Tref r,r1; **//Рабочие указатели.**

Lref p,q; **//Рабочие указатели.**

Lref t; **//Указатель на k-ю вершину частичного решения.**

**int** v; **//Вершина - кандидат на дополнение к частичному решению.**

Boolean Res; **//Флаг клики.**

Boolean Res1;**//Флаг существования ребра.**

**int** i; **//Параметр цикла.**

SearchGraph (X[k-1], &t);

r = t->Trail;

**while** ( r != **NULL** )

{

v = r->Id->Key;

**//Проверим, смежна ли вершина v с вершинами X[1],X[2],...,X[k-1].**

Res = TRUE;

**for** (i=1;i<=k-1;i++)

{

**//Cуществует ли в графе ребро (X[i],v)?**

SearchGraph (v, &p);

SearchGraph (X[i], &q);

r1 = p->Trail;

Res1 = FALSE;

**while** (r1 != **NULL** && !Res1)

**if** ( r1->Id==q ) Res1 = TRUE;

**else** r1 = r1->Next;

Res = (Res && Res1);

}

**if** (!Res) r->Id->Flag = FALSE;

**// --------------------------**

**if** (k==m && Res)

**//Количество вершин в графе равно m, и**

**//вершины X[1],X[2],...,X[k] образуют клику.**

{

**//Вывод клики на экран дисплея.**

**for** (i=1;i<=k-1;i++) cout << X[i] << " ";

cout << v << endl;

}

**else**

**if** ( r->Id->Flag )

{

X[k] = r->Id->Key;

r->Id->Flag = FALSE;

Clique (k+1,m);

r->Id->Flag = TRUE;

}

r = r->Next;

}

}

Текст этой программы можно взять [здесь](http://khpi-iip.mipk.kharkiv.edu/library/datastr/book_sod/kgsu/din113_1.zip).

Мы рассмотрели лишь естественный поиск клик с возвращением, т.е. поиск, в котором не делается никаких попыток упростить дерево поиска. Каждый узел в дереве поиска соответствует полному подграфу графа, и каждое ребро соответствует вершине графа. Заметим, что каждая клика порождается много раз: в общем случае клика размера **k**порождается **k!** раз.

От общего числа шагов порядка **n!** могут спасти (не всегда, но часто) следующие два обстоятельства [2, с.56].

1. Если когда-то уже была найдена **p**-клика, а затем на некоторой ветви процесса образовалась **q**-клика от присоединения (к какой-то другой, вообще говоря, клике) вершины **xj** с номером **j>=0n-p+q**, то развивать эту ветвь не имеет смысла, ибо ни одной клики с числом вершин более **p** она все равно дать не может.
2. Пусть из некоторой сформированной клики **F** мы удаляем последнюю присоединенную вершину **xk** и вместо нее собираемся добавить **xl** с **l>k**; это заведомо не имеет смысла делать, если все вершины клики **F\xk**, смежные с **xl**, смежны также с **xk**.

Весь процесс с учетом этих двух обстоятельств представляет собой частный случай хорошо известного ***метода ветвей и границ***, предлагался в разное время (устно и письменно) многими авторами, и установить приоритет здесь затруднительно.

Ссылка на вариант технического воплощения этого процесса, называемый ***одесским алгоритмом*** есть в монографии [2, с.56].

***Замечания.***

1. *В монографии [3, с.118]* ***кликой*** *называется произвольное подмножество вершин, в котором каждая пара различных вершин соединена ребром графа.*
2. ***Кликовое число графа*** *(****густота*** *или* ***плотность****) - это максимальное число вершин в кликах данного графа [4, с.46]. Тогда, образно говоря, чем более "плотен" граф, тем будет больше кликовое число.*
3. ***Матрицей клик графа*** *называется матрица инциденций вершин графа его кликам, т.е.* ***aij=1****, если вершина* ***j****принадлежит некоторой клике* ***Ki****, и* ***aij=0*** *в противном случае.*
4. *В монографии [5, с.389-396] приведен очень сложный алгоритм порождения* ***всех клик графа****.*
5. ***Мун*** *и* ***Мозер*** *(1965 г.) доказали (см. ссылку в [4, с.70]), что наибольшее число клик, которые могут встретиться в графе с* ***n*** *вершинами, дается соотношениями:*
6. *3n/3 , если n=0(mod 3)*
7. *4\*3(n-4)/3, если n=1(mod 3)*
8. *2\*3(n-2)/3, если n=2(mod 3)*

*Этот результат впервые был получен в 1960 г.* ***Р.Миллером*** *и* ***Д.Малером****, но не был опубликован.*

(1) Харари Ф. Теория графов. - М.: Мир, 1973. - 300 с.

(2) Зыков А.А. Основы теории графов. М.: Наука, 1987. - 384 с.

(3) Липский В. Комбинаторика для программистов: Пер. с польск. - М.: Мир, 1988. - 213 с.

(4) Кристофидес Н. Теория графов. Алгоритмический подход. - М.: Мир, 1978. - 432 с.

(5) Рейнгольд Э., Hивергельт Ю., Део H. Комбинаторные алгоритмы. Теория и практика. - М.: Мир, 1980. - 476 с.

На следующем шаге мы рассмотрим ***независимые множества вершин графа***.

## Независимые множества вершин графа

На этом шаге мы рассмотрим понятие ***числа независимости графа*** и алгоритм нахождения ***наибольшего независимого множества***.

***Определение 1 [1, с.44].***

Рассмотрим неориентированный граф **G=(V,E)**.

***Независимое множество вершин*** (известное также как ***внутренне устойчивое множество***) есть множество вершин графа **G**, такое, что любые две вершины в нем не смежны (никакая пара вершин не соединена ребром).

Независимое множество называется ***максимальным***, когда нет другого независимого множества, в которое оно бы входило.

Например, рассмотрим граф:

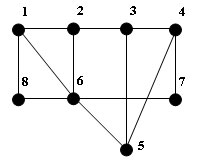


Рис.1. Пример графа

Для данного графа множества вершин

- {7,8,2}, {1,3}, {7,8,2,5} - независимые;

- {1,3,7}, {4,6}, {7,8,2,5} - максимальные.

Следовательно, в рассмотренном графе больше одного независимого множества.

***Определение 2 [1, с.45]***

Если **Q** является семейством всех независимых множеств графа **G**, то число

a(G) = max |S|

S принадлежит Q

называется ***числом независимости графа* G**, а множество **S\***, на котором этот максимум достигается, называется ***наибольшим независимым*** множеством.

Например, для изображенного выше графа семейство максимальных независимых множеств таково: {8,7,2,5}, {1,3,7}, {2,4,8}, {6,4}, {6,3}, {1,4}, {7,5,1}, {3,7,8}.

Наибольшее из этих множеств имеет 4 элемента и, следовательно, **a(G)=4**.

Множество {8,7,2,5} является ***наибольшим независимым множеством***.

Ясно, что понятие, противоположное максимальному независимому множеству, есть ***максимальный полный подграф*** (***клика***).

Совершенно очевидно, что максимальное независимое множество графа **G** соответствует клике графа **G~** и наоборот, где **G~** - дополнение графа **G**.

Это утверждение лежит в основе работы алгоритма, реализованного ниже на языке **C++**.

Пример.

**//Нахождение всех независимых множеств в графе, заданном**

**//структурой Вирта, с использованием клики.**

**#include** <iostream.h>

**#define** TRUE 1

**#define** FALSE 0

**#define** Node 20 **//Максимальное количество вершин в графе.**

**typedef** **int** Boolean;

**typedef** **struct** L \*Lref; **// Тип: указатель на заголовочный узел.**

**typedef** **struct** T \*Tref; **// Тип: указатель на дуговой узел.**

**//Описание типа заголовочного узла.**

**typedef** **struct** L

{

**int** Key; **//Имя заголовочного узла.**

**int** Count; **//Количество предшественников.**

Boolean Flag; **//Флаг посещения узла при обходе.**

Tref Trail; **//Указатель на список смежности.**

Lref Next; **//Указатель на следующий узел в списке заголовочных узлов.**

};

**//Описание типа дугового узла.**

**typedef** **struct** T

{

Lref Id;

Tref Next;

};

**class** Spisok

{

**private**:

Lref Head; **//Указатель на голову списка заголовочных узлов.**

Lref Tail; **//Указатель на фиктивный элемент**

**// в конце списка заголовочных узлов.**

**int** X[20]; **//Результат работы программы.**

**void** SearchGraph (int, Lref \*);

Lref Search(**int**);

**public**:

Spisok() {**//Инициализация списка заголовочных узлов.**

Head = Tail = **new** (L); }

Lref GetHead() { **return** Head; }

Lref GetTail() { **return** Tail; }

**void** MakeGraph ();

**void** PrintGraph ();

**void** AddGraph(int, int);

**void** DeleteGraph(int, int);

**void** Clique (int, int);

**void** X1 (Lref t) {X[1] = t->Key;};

};

**void** main ()

{

Spisok A;

Lref t; **//Рабочий указатель для перемещения**

**// по списку заголовочных звеньев.**

Lref t1;

**int** n=0;

**//Построение графа и вывод его структуры Вирта.**

A.MakeGraph ();

A.PrintGraph (); cout<<endl;

**//Инициализация и подсчет количества вершин графа.**

t = A.GetHead();

**while** (t!=A.GetTail())

{ n++; t = (\*t).Next; }

**// ------------------------------------**

**//Нахождение всех клик в графе Head.**

**for** (**int** i=3;i<=n;i++)

{

**//i - количество вершин в клике (i<=3).**

cout << "Перечислим все клики длины " << i << endl;

t = A.GetHead();

**while** (t!=A.GetTail())

{

**//Инициализация.**

t1 = A.GetHead();

**while** (t1!=A.GetTail())

{ t1->Flag = TRUE; t1 = t1->Next; }

A.X1(t); t->Flag = FALSE;

**//Отыскание клики с i вершинами, "начинающейся" в вершине t.**

A.Clique (2,i); t = t->Next;

}

}

}

Lref Spisok::Search (**int** w)

**//Функция возвращает указатель на заголовочный узел**

**//ключом w. Если узел отсутствует, то функция возвращает NULL .**

{

Lref h = Head;

(\*Tail).Key = w; **//Поиск "с барьером".**

**while** ((\*h).Key!=w) h = (\*h).Next;

**if** (h==Tail) **//В списке заголовочных узлов нет узла с ключом w.**

h = **NULL**;

**return** h;

}

**void** Spisok::AddGraph (**int** x,int y)

**//Добавление дуги (x,y) (если ее не было!) к структуре**

**//Вирта, соответствующей ориентированному графу Head.**

{

Lref p,q; **//Рабочие указатели.**

Tref t,r; **//Рабочие указатели.**

Boolean Res; **//Флаг наличия в графе данной дуги.**

**//Определим, существует ли в графе дуга (x,y)?**

SearchGraph (x,&p); SearchGraph (y,&q);

r = (\*p).Trail; Res = FALSE;

**while** ((r!=**NULL**)&&(!Res))

**if** ((\*r).Id==q) Res = TRUE; **else** r = (\*r).Next;

**if** (!Res)

{ **//Если дуга отсутствует, то поместим её в граф.**

t = **new** (T); (\*t).Id = q; (\*t).Next = (\*p).Trail;

(\*p).Trail = t; (\*q).Count++;

}

}

**void** Spisok::DeleteGraph (**int** x,int y)

**//Функция возвращает указатель Head на структуру**

**//Вирта, соответствующую ориентированному графу**

**//и полученную удалением дуги (x,y).**

{

Lref p,q; **//Рабочие указатели.**

Tref t,r; **//Рабочие указатели.**

Boolean Res; **//Флаг наличия в графе данной дуги.**

**//Определим, существует ли в графе дуга (x,y)?**

p=Search (x); q= Search (y);

**if** ((p!=**NULL**)&&(q!=**NULL**))

{ **//Вершины x и y в графе есть.**

r = (\*p).Trail; Res = FALSE;

**while** ((r!=**NULL**)&&(!Res))

**if** ((\*r).Id==q) Res = TRUE;

**else** r = (\*r).Next;

**if** (Res) **//Если дуга существует, то удалим её.**

**if** (r==(\*p).Trail)

{ (\*p).Trail = (\*(\*p).Trail).Next; **delete** r; (\*q).Count--; }

**else**

{

t = (\*p).Trail;

**while** ((\*t).Next!=r) t = (\*t).Next;

(\*t).Next = (\*(\*t).Next).Next; **delete** r; (\*q).Count--; }

}

}

**void** Spisok::SearchGraph (**int** w, Lref \*h)

**//Функция возвращает указатель на заголовочный узел**

**//с ключом w в графе, заданном структурой Вирта с указателем Head.**

{

\*h = Head; (\*Tail).Key = w;

**while** ((\*\*h).Key!=w) \*h = (\*\*h).Next;

**if** (\*h==Tail)

**//В списке заголовочных узлов нет узла с ключом w.**

**//Поместим его в конец списка Head.**

{ Tail = **new** (L); (\*\*h).Count = 0;

(\*\*h).Trail = **NULL**; (\*\*h).Next = Tail; }

}

**void** Spisok::MakeGraph ()

**//Функция возвращает указатель Head на структуру**

**//Вирта, соответствующую ориентированному графу.**

{

**int** x,y;

Lref p,q; **//Рабочие указатели.**

Tref t,r; **//Рабочие указатели.**

Boolean Res; **//Флаг наличия дуги.**

cout<<"Вводите начальную вершину дуги: ";

cin>>x;

**while** (x!=0)

{

cout<<"Вводите конечную вершину дуги: "; cin>>y;

**//Определим, существует ли в графе дуга (x,y)?**

SearchGraph (x, &p); SearchGraph (y,&q);

r = (\*p).Trail; Res = FALSE;

**while** ((r!=**NULL**)&&(!Res))

**if** ((\*r).Id==q) Res = TRUE;

**else** r = (\*r).Next;

**if** (!Res) **//Если дуга отсутствует, то поместим её в граф.**

{ t = **new** (T); (\*t).Id = q;

(\*t).Next = (\*p).Trail; (\*p).Trail = t; (\*q).Count++; }

cout<<"Вводите начальную вершину дуги: "; cin>>x;

}

}

**void** Spisok::PrintGraph ()

**//Вывод структуры Вирта, заданной указателем**

**//Head и соответствующей ориентированному графу.**

{

Lref p; **//Рабочий указатель.**

Tref q; **//Рабочий указатель.**

p = Head;

**while** (p!=Tail)

{

cout<< (\*p).Key << "("; q = (\*p).Trail;

**while** (q!=**NULL**)

{ cout<<(\*(\*q).Id).Key; q = (\*q).Next; }

cout<<")"; p = (\*p).Next; cout<<" ";

}

}

**void** Spisok::Clique (**int** k, **int** m)

**//Нахождение всех клик, содержащих m вершин, в графе,**

**//заданном структурой Вирта с указателем Head,**

**//k - количество вершин в частичном решении.**

{

Tref r,r1; **//Рабочие указатели.**

Lref p,q; **//Рабочие указатели.**

Lref t; **//Указатель на k-ю вершину частичного решения.**

**int** v; **//Вершина - кандидат на дополнение к частичному решению.**

Boolean Res; **//Флаг клики.**

Boolean Res1;**//Флаг существования ребра.**

**int** i; **//Параметр цикла.**

SearchGraph (X[k-1], &t);

r = t->Trail;

**while** ( r != **NULL** )

{

v = r->Id->Key;

**//Проверим, смежна ли вершина v с вершинами X[1],X[2],...,X[k-1].**

Res = TRUE;

**for** (i=1;i<=k-1;i++)

{

**//Cуществует ли в графе ребро (X[i],v)?**

SearchGraph (v, &p);

SearchGraph (X[i], &q);

r1 = p->Trail;

Res1 = FALSE;

**while** (r1 != **NULL** && !Res1)

**if** ( r1->Id==q ) Res1 = TRUE;

**else** r1 = r1->Next;

Res = (Res && Res1);

}

**if** (!Res) r->Id->Flag = FALSE;

**// --------------------------**

**if** (k==m && Res)

**//Количество вершин в графе равно m, и**

**//вершины X[1],X[2],...,X[k] образуют клику.**

{

**//Вывод клики на экран дисплея.**

**for** (i=1;i<=k-1;i++) cout << X[i] << " ";

cout << v << endl;

}

**else**

**if** ( r->Id->Flag )

{

X[k] = r->Id->Key;

r->Id->Flag = FALSE;

Clique (k+1,m);

r->Id->Flag = TRUE;

}

r = r->Next;

}

}

Текст этой программы можно взять [здесь](http://khpi-iip.mipk.kharkiv.edu/library/datastr/book_sod/kgsu/din114_1.zip).

***Замечания.***

1. *Пусть нам необходимо отыскать "специальное" множество* ***M*** *ребер графа* ***G*** *такое, что в* ***M*** *не должно быть смежных ребер. Множество* ***M*** *называют* ***паросочетанием****, а множество* ***M\**** *- с наибольшей мощностью - является* ***наибольшим паросочетанием****, его можно называть также* ***наибольшим независимым множеством ребер*** *[1, с.66].*

*Связь между наибольшими независимыми множествами вершин и наибольшими паросочетаниями устанавливается с помощью* ***реберных графов****.*

1. *[2, с.73]. Одна из чисто житейских интерпретаций понятия независимости состоит в следующем. Определенный человек желает устроить юбилей с максимальным числом гостей из своих знакомых. Стремясь сделать юбилейный вечер приятным, он должен организовать все так, чтобы на этом вечере присутствовали люди, симпатизирующие друг другу. Так как отношение "симпатии" не является транзитивным, то ему для достижения цели придется находить число независимости* ***графа своих знакомых****.*

*Этот граф устроен следующим образом. Вершины его - знакомые юбиляра. Две вершины смежны, если соответствующие знакомые друг другу не симпатизируют. Нетрудно понять, что число независимости этого графа и представляет тот самый максимальный контингент приглашенных, который может себе позволить юбиляр.*

1. *Рассмотрим следующую шахматную задачу. Сколько ферзей можно установить на шахматной жоске, чтобы они не "били" друг друга? Нетрудно убедиться, что искомое число равно 8, что соответствует утверждению о числе независимости* ***графа шахматной доски G****. Ясно, что аналогичные задачи могут быть сформулированы и для других шахматных фигур.*
2. *[2, с.74]. Пусть* ***Gn*** *- граф "делимости" натуральных чисел. Вершины* ***Gn*** *- это числа 1,2,...,n, и две вершины* ***i, j****смежны только в том случае, когда их наибольший общий делитель больше единицы. На рисунке изображен граф* ***G8****:*

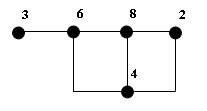
**

Рис.2. Граф **G8**

*Нетрудно показать, что справедливо следущее равенство:*

*a(Gn)=p(n),*

*где функция* ***p(n)*** *- число простых чисел, не превосходящих* ***n****. Полученное соотношение показывает, что нахождение числа независимости в общем случае не является простой задачей, уже хотя бы потому, что для частного вида графа она эквивалентна традиционно трудной задаче вычисления значения функции* ***p(n)****.*

1. *В работе [3] показано, что "почти все"* ***n****-вершинные графы имеют число независимости, лежащее в следующих границах:*
2. *[2(log2n-log2log2n)]<=a(G)<=[2(log2n-log2log2n)]+3*

(1) Кристофидес Н. Теория графов. Алгоритмический подход. - М.: Мир, 1978. - 432 с.

(2) Компьютер и задачи выбора. - М.: Наука, 1989. - 208 с.

(3) Коршунов А.Д. Основные свойства случайных графов с большим числом вершин и ребер // УМН, 1985, т.40, вып.1 (241), с.107-173.

Со следующего шага мы начнем знакомиться с ***раскрасками***.

## Раскраски

На этом шаге мы приведем ***общие сведения о раскрасках***.

Разнообразные задачи, возникающие при планировании производства, составлении графиков осмотра, хранении и транспортировке товаров и т.д. могут быть представлены часто как задачи теории графов, тесно связанные с так называемой "***задачей раскраски***".

Пусть рассматриваемые графы являются неориентированными и не имеют петель.

***Определение [1, с.73].***

Граф **G** называют **r-*хроматическим***, если его вершины могут быть раскрашены с использованием **r** цветов (красок) так, что не найдется двух смежных вершин одного цвета.

Наименьшее число **r**, такое, что граф **G** является **r**-хроматическим, называется ***хроматическим числом*** графа **G** и обозначается **g(G)**.

Например, полный **n**-вершинный граф **G** имеет хроматическое число, равное **n**, а всякое дерево имеет хроматическое число равное двум.

Задача нахождения хроматического числа графа называется ***задачей о раскраске (задачей раскраски) графа***.

Соответствующая хроматическому числу раскраска вершин разбивает множество вершин графа на **r** подмножеств, каждое из которых содержит вершины одного цвета. Эти множества являются независимыми, поскольку в пределах одного множества нет двух смежных вершин.

Вообще говоря, хроматическое число графа нельзя найти, зная только число вершин и число ребер графа. При известных величинах **n** (число вершин), **m** (число ребер) и **d1, d2, ..., dn** (степени вершин графа) можно получить верхнюю и нижнюю оценки для хроматического числа графа.

Вначале отметим, что поскольку между кликами графа **G** и максимальными независимыми множествами дополнительного графа **G~** существует взаимно однозначное соответствие, то справедливы равенства

r(G)= a(G~) и r(G~0)=a(G),

где **a(G)** - ***число независимости*** графа, **r(G)** - ***кликовое число*** графа **G**.

Теперь, поскольку по крайней мере **r(G)** цветов требуется для раскраски соответствующей клики графа **G** (той самой клики, которая "определяет" кликовое число графа **G**), что **r(G)** является ***нижней оценкой хроматического числа***, т.е. **g(G)>=r(G)**.

***У.Татт*** в 1954 г. показал, что можно построить граф **G** с кликовым числом равным 2, который будет иметь произвольно большое заданное значение хроматического числа (***граф Татта***).

Поскольку **a(G)** равно мощности наибольшего множества попарно несмежных вершин графа **G**, то оно совпадает также с мощностью наибольшего множества вершин в **G**, которые могут быть окрашены в один цвет, и, следовательно,

g(G) >= [n/a(G)]

где **n** - число вершин графа **G**, а **[x]** - целая часть числа **x**.

Нижние оценки хроматического числа безусловно более интересны, чем верхние, поскольку (если они достаточно близки к истинному значению) они могут быть использованы в процедуре вычисления **g(G)**, включающей дерево поиска. В то же время верхние оценки хроматического числа подобного применения не находят.

Тем не менее в литературе приводятся формулы для вычисления верхних оценок хроматического числа; так ***Р.Бруксом*** предложена следующая легко вычисляемая оценка:

g(G) <= 1 + max [d(vi)+1]

vi принадлежит V

где **d(vi)** - степень вершины **vi**.

Приведем важную для построения алгоритма раскраски теорему.

***Теорема [1, с.80].***

Если граф является **r**-хроматическим, то он может быть раскрашен с использованием **r** (или меньшего числа) красок с помощью следующей процедуры: сначала в один цвет окрашивается некоторое максимальное независимое множество **S1[G]**, затем окрашивается в следующий цвет множество **S1[X-S1[G]]** и т.д. до тех пор, пока не будут раскрашены все вершины.

Раскраску указанного в теореме вида будем называть ***оптимальной независимой раскраской***.

Приведенный ниже пример показывает шаги алгоритма, описанного в теореме.

Максимальным независимым множеством вершин для графа

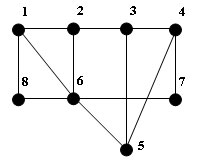


Рис.1. Пример графа

является {8,7,2,5}. Удаляем данные вершины из графа, предварительно присвоив им цвет 1:

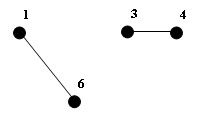


Рис.2. Результат удаления

Максимальными независимыми множествами вершин для полученного графа будут {1,3}, {1,4}, {6,3}, {6,4}. Удалим из графа, например, вершины 1 и 4, присвоив им цвет 2. Оставшиеся в графе вершины 3 и 6 получают цвет 3.

Изобразим результат раскраски:

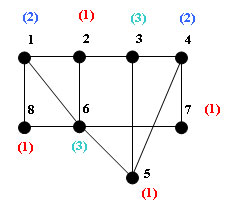


Рис.3. Результат раскраски

(1) Кристофидес Н. Теория графов. Алгоритмический подход. - М.: Мир, 1978. - 432 с.

На следующем шаге мы рассмотрим ***алгоритмы раскраски***.

## Алгоритмы раскраски графа

На этом шаге мы рассмотрим ***алгоритмы закраски графа***.

Задачи определения хроматического числа и построения минимальной раскраски произвольного графа являются очень сложными. С одной стороны, не известны алгоритмы их решения, сложность которых есть некоторая фиксированная степень от длины записи условий задачи (так называемые ***полиномиальные алгоритмы***). С другой стороны, нигде явно не выражены те потери, которые мы несем от отсутствия таких алгоритмов [1, с.70].

Разумеется, есть лишь конечное число ситуаций, которые надо рассмотреть, чтобы найти хроматическое число **n**-вершинного графа. Действительно, всего имеется **kn** различных способов раскраски графа **G** в **k** цветов. Перебрав их, мы либо найдем правильную раскраску **G** в **k** цветов, либо убедимся, что таковой не существует. Ясно, что это самый примитивный способ решения задачи. Имеется целый ряд значительно более совершенных методов, использующих как оценки, так и другие соображения, призванные сократить перебор.

Рассмотрим простой ***алгоритм построения правильной раскраски*** [2, с.237], в ряде случаев приводящий к раскраскам, близким к минимальным.

***Алгоритм последовательной раскраски***

1. Произвольной вершине **v1** графа **G** припишем цвет 1.
2. Если вершины **v1, v2, ...,vi** раскрашены **l** цветами 1, 2, ..., **l**, **l<=i**, то новой произвольно взятой вершине **vi+1**припишем минимальный цвет, не использованный при раскраске смежных вершин.

Приведем реализацию алгоритма, осуществляющего ***последовательную раскраску вершин графа при помощи обхода графа в глубину***.

Пример 1.

**//Последовательная раскраска вершин графа при помощи**

**//обхода графа в глубину.**

**//Граф представлен структурой Вирта.**

**#include** <iostream.h>

**#define** TRUE 1

**#define** FALSE 0

**typedef** **int** Boolean;

**typedef** **struct** L \*Lref; **// Тип: указатель на заголовочный узел.**

**typedef** **struct** T \*Tref; **// Тип: указатель на дуговой узел.**

**//Описание типа заголовочного узла.**

**typedef** **struct** L

{

**int** Key; **//Имя заголовочного узла.**

**int** Color; **//Цвет раскраски.**

**int** Count; **//Количество предшественников.**

Boolean Flag; **//Флаг посещения узла при обходе.**

Tref Trail; **//Указатель на список смежности.**

Lref Next; **//Указатель на следующий узел в списке заголовочных узлов.**

};

**//Описание типа дугового узла.**

**typedef** **struct** T

{

Lref Id;

Tref Next;

};

**class** Spisok

{

**private**:

Lref Head; **//Указатель на голову списка заголовочных узлов.**

Lref Tail; **//Указатель на фиктивный элемент**

**// в конце списка заголовочных узлов.**

**int** MSet[256]; **//Вспомогательное множество, содер-**

**//жащее 0,1,2...,n.**

**void** SearchGraph (int, Lref \*);

**public**:

Spisok() {**//Инициализация списка заголовочных узлов.**

Head = Tail = **new** (L); }

Lref GetHead() { **return** Head; }

Lref GetTail() { **return** Tail; }

**void** MakeGraph ();

**void** PrintGraph ();

**void** Color (Lref, **int**);

**void** Postr (**int**);

};

**void** main ()

{

Spisok A;

Lref t; **//Рабочий указатель для перемещения**

**// по списку заголовочных звеньев.**

**int** n=0; **//Количество вершин в графе.**

**//Построение графа и вывод его структуры Вирта.**

A.MakeGraph ();

A.PrintGraph (); cout<<endl;

**//Раскраска с помощью рекурсивного обхода графа в глубину.**

**//Инициализация.**

t = A.GetHead(); **//Установлен рабочий указатель.**

**while** (t!=A.GetTail())

{ t->Flag = TRUE; t->Color = 0;

n++; t = (\*t).Next; }

**// ------------------------------------**

**//Построение вспомогательного множества MSet.**

A.Postr(n);

cout << "Результат раскраски: ";

A.Color (A.GetHead(),n);

cout << endl;

}

**void** Spisok::Postr (**int** n)

**//Построение вспомогательного множества MSet.**

{

**for** (**int** i=0;i<256;i++)

MSet[i] = (i<=n)?1:0;

}

**void** Spisok::SearchGraph (**int** w, Lref \*h)

**//Функция возвращает указатель на заголовочный узел**

**//с ключом w в графе, заданном структурой Вирта с указателем Head.**

{

\*h = Head; (\*Tail).Key = w;

**while** ((\*\*h).Key!=w) \*h = (\*\*h).Next;

**if** (\*h==Tail)

**//В списке заголовочных узлов нет узла с ключом w.**

**//Поместим его в конец списка Head.**

{ Tail = **new** (L); (\*\*h).Count = 0;

(\*\*h).Trail = **NULL**; (\*\*h).Next = Tail; }

}

**void** Spisok::MakeGraph ()

**//Функция возвращает указатель Head на структуру**

**//Вирта, соответствующую ориентированному графу.**

{

**int** x,y;

Lref p,q; **//Рабочие указатели.**

Tref t,r; **//Рабочие указатели.**

Boolean Res; **//Флаг наличия дуги.**

cout<<"Вводите начальную вершину дуги: ";

cin>>x;

**while** (x!=0)

{

cout<<"Вводите конечную вершину дуги: "; cin>>y;

**//Определим, существует ли в графе дуга (x,y)?**

SearchGraph (x, &p); SearchGraph (y,&q);

r = (\*p).Trail; Res = FALSE;

**while** ((r!=**NULL**)&&(!Res))

**if** ((\*r).Id==q) Res = TRUE;

**else** r = (\*r).Next;

**if** (!Res) **//Если дуга отсутствует, то поместим её в граф.**

{ t = **new** (T); (\*t).Id = q;

(\*t).Next = (\*p).Trail; (\*p).Trail = t; (\*q).Count++; }

cout<<"Вводите начальную вершину дуги: "; cin>>x;

}

}

**void** Spisok::PrintGraph ()

**//Вывод структуры Вирта, заданной указателем**

**//Head и соответствующей ориентированному графу.**

{

Lref p; **//Рабочий указатель.**

Tref q; **//Рабочий указатель.**

p = Head;

**while** (p!=Tail)

{

cout<< (\*p).Key << "("; q = (\*p).Trail;

**while** (q!=**NULL**)

{ cout<<(\*(\*q).Id).Key; q = (\*q).Next; }

cout<<")"; p = (\*p).Next; cout<<" ";

}

}

**void** Spisok::Color (Lref r, **int** n)

**//Последовательная раскраска графа при помощи**

**//рекурсивного обхода графа в глубину.**

**//r - указатель на структуру Вирта.**

**//MSet - глобальное множество.**

**//n - количество вершин в графе.**

{

Tref t,t1;

**int** i; **//Параметр цикла.**

Boolean Fl;

t = r->Trail; r->Flag = FALSE;

**//Сейчас мы находимся в вершине r->Key.**

**//Исключим цвета всех вершин, смежных вершине**

**//r->Key, из множества MSet.**

t1 = t;

**while** ( t1 != **NULL** )

{ MSet[t1->Id->Color] = 0; t1 = t1->Next; }

**//Выбор цвета вершины: это "первый" элемент множества MSet.**

Fl = TRUE; i = 1;

**while** ( Fl )

**if** ( MSet[i] ) Fl = FALSE; **else** i++;

r->Color = i; **//Цвет присвоен!**

cout << "(" << r->Key << "," << r->Color << ") ";

**//Восстановление вспомогательного множества MSet.**

**for** (i=0;i<256;MSet[i++]=0);

**for** (i=0;i<=n;MSet[i++]=1);

**// -------------**

**while** ( t != **NULL** )

{

**if** ( t->Id->Flag ) Color (t->Id,n);

t = t->Next;

}

}

Текст этой программы можно взять [здесь](http://khpi-iip.mipk.kharkiv.edu/library/datastr/book_sod/kgsu/din116_1.zip).

Приведем реализацию алгоритма, осуществляющего ***последовательную раскраску вершин графа при помощи обхода графа в ширину***.

**//Последовательная раскраска графа при помощи**

**//обхода графа в ширину.**

**//Граф представлен структурой Вирта.**

**#include** <iostream.h>

**#define** TRUE 1

**#define** FALSE 0

**typedef** **int** Boolean;

**typedef** **struct** Leader \*Lref; **// Тип: указатель на заголовочный узел.**

**typedef** **struct** Trailer \*Tref; **// Тип: указатель на дуговой узел.**

**//Описание типа заголовочного узла.**

**typedef** **struct** Leader

{

**int** Key; **//Имя заголовочного узла.**

**int** Count; **//Количество предшественников.**

**int** Color; **//Цвет раскраски.**

Boolean Flag; **//Флаг посещения узла при обходе.**

Tref Trail; **//Указатель на список смежности.**

Lref Next; **//Указатель на следующий узел в списке заголовочных узлов.**

};

**//Описание типа дугового узла.**

**typedef** **struct** Trailer

{

Lref Id;

Tref Next;

};

**//Описание типа узла очереди.**

**typedef** Lref TipElement; **//Указатель на звено заголовочного списка.**

**typedef** **struct** Zveno \*svqz;

**typedef** **struct** Zveno

{

TipElement Element; **//Указатель на список смежности.**

svqz Sled;

};

**class** Spisok

{

**private**:

Lref Head; **//Указатель на голову списка заголовочных узлов.**

Lref Tail; **//Указатель на фиктивный элемент**

**// в конце списка заголовочных узлов.**

**int** MSet[256]; **//Вспомогательное множество, содер-**

**//жащее 0,1,2...,n.**

**void** Udalenie\_A (svqz \*, svqz \*, TipElement \*);

**void** Dobavlenie (svqz \*, svqz \*, TipElement);

**void** SearchGraph (int, Lref \*);

**public**:

Spisok() {**//Инициализация списка заголовочных узлов.**

Head = Tail = **new** (Leader); }

Lref GetHead() { **return** Head; }

Lref GetTail() { **return** Tail; }

**void** MakeGraph ();

**void** PrintGraph ();

**void** Color (Lref, **int**);

**void** Postr (**int**);

};

**void** main ()

{

Spisok A;

Lref t; **//Рабочий указатель для перемещения**

**// по списку заголовочных звеньев.**

**int** n=0; **//Количество вершин в графе.**

**//Построение графа и вывод его структуры Вирта.**

A.MakeGraph ();

A.PrintGraph (); cout<<endl;

**//Раскраска с помощью рекурсивного обхода графа в ширину.**

**//Инициализация.**

t = A.GetHead(); **//Установлен рабочий указатель.**

**while** (t!=A.GetTail())

{ t->Flag = TRUE; t->Color = 0;

n++; t = (\*t).Next; }

**// ------------------------------------**

**//Построение вспомогательного множества MSet.**

A.Postr(n);

cout << "Результат раскраски: ";

A.Color (A.GetHead(),n);

cout << endl;

}

**void** Spisok::Postr (**int** n)

**//Построение вспомогательного множества MSet.**

{

**for** (**int** i=0;i<256;i++)

MSet[i] = (i<=n)?1:0;

}

**void** Spisok::SearchGraph (**int** w, Lref \*h)

**//Функция возвращает указатель на заголовочный узел**

**//с ключом w в графе, заданном структурой Вирта с указателем Head.**

{

\*h = Head; (\*Tail).Key = w;

**while** ((\*\*h).Key!=w) \*h = (\*\*h).Next;

**if** (\*h==Tail)

**//В списке заголовочных узлов нет узла с ключом w.**

**//Поместим его в конец списка Head.**

{ Tail = **new** (Leader); (\*\*h).Count = 0;

(\*\*h).Trail = **NULL**; (\*\*h).Next = Tail; }

}

**void** Spisok::MakeGraph ()

**//Функция возвращает указатель Head на структуру**

**//Вирта, соответствующую ориентированному графу.**

{

**int** x,y;

Lref p,q; **//Рабочие указатели.**

Tref t,r; **//Рабочие указатели.**

Boolean Res; **//Флаг наличия дуги.**

cout<<"Вводите начальную вершину дуги: ";

cin>>x;

**while** (x!=0)

{

cout<<"Вводите конечную вершину дуги: "; cin>>y;

**//Определим, существует ли в графе дуга (x,y)?**

SearchGraph (x, &p); SearchGraph (y,&q);

r = (\*p).Trail; Res = FALSE;

**while** ((r!=**NULL**)&&(!Res))

**if** ((\*r).Id==q) Res = TRUE;

**else** r = (\*r).Next;

**if** (!Res) **//Если дуга отсутствует, то поместим её в граф.**

{ t = **new** (Trailer); (\*t).Id = q;

(\*t).Next = (\*p).Trail; (\*p).Trail = t; (\*q).Count++; }

cout<<"Вводите начальную вершину дуги: "; cin>>x;

}

}

**void** Spisok::PrintGraph ()

**//Вывод структуры Вирта, заданной указателем**

**//Head и соответствующей ориентированному графу.**

{

Lref p; **//Рабочий указатель.**

Tref q; **//Рабочий указатель.**

p = Head;

**while** (p!=Tail)

{

cout<<(\*p).Key<<"("; q = (\*p).Trail;

**while** (q!=**NULL**)

{ cout<<(\*(\*q).Id).Key; q = (\*q).Next; }

cout<<")"; p = (\*p).Next; cout<<" ";

}

}

**void** Spisok::Dobavlenie (svqz \*L, svqz \*R, TipElement elem)

**//Добавление элемента elem в очередь, заданную указателями L и R.**

{

svqz K = **new** (Zveno);

K->Element = elem; K->Sled = **NULL**;

**if** (\*L==**NULL**)

{ (\*L) = K; (\*R) = K; }

**else** { (\*R)->Sled = K; (\*R) = K; }

}

**void** Spisok::Udalenie\_A (svqz \*L, svqz \*R, TipElement \*A)

**//Удаление элемента из очереди, заданной указателями L и R и**

**//помещение удаленного элемента в переменную A.**

{

svqz q;

**if** ((\*L)!=**NULL**)

**if** ((\*L)->Sled!=**NULL**)

{

(\*A) = (\*L)->Element; q = (\*L);

(\*L) = (\*L)->Sled; **delete** q;

}

**else** {

(\*A) = (\*L)->Element; **delete** \*L;

(\*L) = (\*R) = **NULL**;

}

}

**void** Spisok::Color (Lref H, **int** n)

**//Последовательная раскраска графа при помощи обхода графа**

**//в ширину, начиная с вершины, заданной указателем H**

**//(нерекурсивный обход).**

{

svqz L; **// Указатель на начало очереди.**

svqz R; **// Указатель на конец очереди.**

Lref p; **// Рабочий указатель.**

Tref t; **// Рабочий указатель.**

Tref t1;

**int** i; **// Параметр цикла.**

Boolean Fl;

L = R = **NULL**; **// Создание пустой очереди.**

Dobavlenie (&L,&R,H); H->Flag = FALSE;

**while** ( L != **NULL** )

{ **//Пока очередь не пуста...**

Udalenie\_A (&L,&R,&p);

t = p->Trail;

**//Сейчас мы находимся в вершине r->Key.**

**//Исключим цвета всех вершин, смежных вершине**

**//r->Key, из множества MSet.**

t1 = t;

**while** ( t1 != **NULL** )

{ MSet[t1->Id->Color] = 0; t1 = t1->Next; }

**//Выбор цвета вершины - "первого" элемента множества MSet.**

Fl = TRUE; i = 1;

**while** ( Fl )

**if** ( MSet[i] ) Fl = FALSE; **else** i++;

p->Color = i; **//Цвет присвоен!**

cout << "(" << p->Key << "," << p->Color << ") ";

**//Восстановление вспомогательного множества MSet.**

**for** (i=0;i<256;MSet[i++]=0);

**for** (i=0;i<=n;MSet[i++]=1);

**while** ( t != **NULL** )

{

**if** ( t->Id->Flag )

{

Dobavlenie (&L,&R,t->Id);

t->Id->Flag = FALSE;

}

t = t->Next;

}

}

}

Текст этой программы можно взять [здесь](http://khpi-iip.mipk.kharkiv.edu/library/datastr/book_sod/kgsu/din116_2.zip).

Пример 3.

**//Последовательная раскраска графа, представленного с помощью**

**//модифицированных списков смежности.**

**#include** <iostream.h>

**#include** <conio.h>

**#define** TRUE 1

**#define** FALSE 0

**#define** XRY 8 **//Количество вершин графа.**

**typedef** **int** Boolean;

**typedef** **struct** zveno \*svqz;

**typedef** **struct** zveno

{

**int** Key; **// Вершина графа.**

svqz Up; **// Указатель на смежную вершину.**

svqz Sled;**// Указатель на следующую смежную вершину.**

};

**class** Spisok

{

**private**:

svqz Beg[XRY+1]; **//Массив указателей на вершины.**

**void** Add\_Ver (**int**);

**int** Find (int, int, svqz \*);

**void** Add\_duga (int, int);

**void** Make (int, int);

**void** Delinenok (int, int);

**void** Del\_Duga (int, int);

**void** Del\_Ver (**int**);

**int** Find\_Color (int, int, svqz \*);

**public**:

Spisok();

**void** Init\_Graph ();

**void** Make\_Graph ();

**void** Print\_Graph ();

**void** Color ();

**void** Print\_Color ();

};

**void** main ()

{

Spisok A;

**//Инициализация графа.**

A.Init\_Graph ();

**//Построение графа.**

A.Make\_Graph ();

**//Вывод графа.**

A.Print\_Graph ();

getch();

**//Последовательная раскраска графа, представленного**

**//модифицированными списками смежности.**

A.Color ();

A.Print\_Color ();

getch();

}

Spisok::Spisok()

{

**for** ( **int** i=0; i<=XRY;Beg[i++] = **NULL** );

}

**void** Spisok::Add\_Ver (**int** x)

**//Функция создания вершины x.**

{

svqz Ukaz = **new** (zveno);

Ukaz->Sled = **NULL**;

Beg[x] = Ukaz;

}

**void** Spisok::Init\_Graph ()

**//Функция инициализации модифицированных списков смежности.**

{

**for** (**int** i=1;i<=XRY;i++) Add\_Ver (i);

}

**int** Spisok::Find (**int** x, **int** y, svqz \*UkZv)

**//Функция проверки смежности вершин y и x. Функция возвращает**

**//TRUE, если вершина x смежна с вершиной y. Указатель \*UkZv,**

**//возвращает указатель на место y в списке смежности x. Если**

**//вершина x не смежна с вершиной y, то UkZv есть NULL.**

{

svqz Ukaz;

Ukaz = Beg[x]->Sled;

**while** (Ukaz != **NULL** && Ukaz->Key != y)

Ukaz = Ukaz->Sled;

\*UkZv = Ukaz;

**return** ( Ukaz != **NULL** );

}

**void** Spisok::Add\_duga (**int** x, **int** y)

**//Функция создания дуги (x,y).**

{

svqz Ukaz = **new** (zveno);

Ukaz->Sled = Beg[x]->Sled; Ukaz->Key = y;

Beg[x]->Sled = Ukaz;

}

**void** Spisok::Make (**int** x, **int** y)

**//Функция создания ребра {x,y}.**

{

svqz Ukaz;

**if** ( !Find(x,y,&Ukaz) )

{

Add\_duga (x,y);

**if** ( x != y ) Add\_duga (y,x);

Beg[x]->Sled->Up = Beg[y];

Beg[y]->Sled->Up = Beg[x];

}

}

**void** Spisok::Make\_Graph ()

**//Функция построения модифицированных списков смежности.**

{

**int** f;

**for** (**int** i=1;i<=XRY;i++)

{

cout << "Введите вершины, смежные с " << i << "-й вершиной: ";

cin >> f;

**while** (f!=0)

{

Make (i,f);

cout << " ";

cin >> f;

}

cout << endl;

}

}

**void** Spisok::Delinenok (**int** x, **int** y)

**//Функция удаления дуги (x,y).**

{

svqz Ukaz;

svqz Ukazlenok;

Ukazlenok = Beg[x]; Ukaz = Beg[x]->Sled;

**while** (Ukaz != **NULL** && Ukaz->Key != y)

{ Ukazlenok = Ukaz; Ukaz = Ukaz->Sled; }

**if** ( Ukaz != **NULL** )

{

Ukazlenok->Sled = Ukaz->Sled;

**delete** Ukaz;

}

}

**void** Spisok::Del\_Duga (**int** x, **int** y)

**//Функция удаления ребра {x,y}.**

{

Delinenok (x,y); Delinenok (y,x);

}

**void** Spisok::Del\_Ver (**int** x)

**//Функция удаления вершины x.**

{

svqz Ukaz;

svqz Ukazlenok;

**for** (**int** i=1;i<=XRY;i++) Delinenok (i,x);

Ukaz = Beg[x]; Beg[x] = **NULL**;

**while** ( Ukaz != **NULL** )

{

Ukazlenok = Ukaz->Sled;

**delete** Ukaz; Ukaz = Ukazlenok;

}

}

**void** Spisok::Print\_Graph ()

**//Функция вывода содеpжимого модифицированных**

**//списков смежности.**

{

svqz UkZv;

**for** (**int** i=1;i<=XRY;i++)

{

**if** ( Beg[i] != **NULL** )

{

UkZv = Beg[i]->Sled;

cout << i << " - ";

**while** ( UkZv != **NULL** )

{

cout << " " << UkZv->Key;

UkZv = UkZv->Sled;

}

}

cout << endl;

}

}

**int** Spisok::Find\_Color (**int** x, **int** c, svqz \*UkZv)

**//Функция выявления возможности раскраски вершины x цветом c.**

**//Функция возвращает TRUE, если вершину x можно раскрасить.**

**//Указатель \*UkZv, возвращает указатель на вершину, смежную с x**

**//и раскрашенную цветом c. Если вершину x можно раскрасить, то**

**//\*UkZv есть NULL.**

{

**int** i = 1;

**while** (!(Find(x,i,&(\*UkZv)) &&

Beg[i]->Key==c) &&

i != x) i++;

**return** (i==x);

}

**void** Spisok::Color ()

**//Функция последовательной раскраски графа, заданного**

**//модифицированными списками смежности Beg.**

{

**int** i = 1;

**int** c = 1;

svqz UkZv;

**while** (Beg[i] == **NULL** && i<=XRY) i++;

**if** ( i != XRY )

{

Beg[i]->Key = c;

i++;

**while** (i<=XRY)

**if** ( Beg[i] != **NULL** )

{

c = 1;

**while** (!Find\_Color(i,c,&UkZv)) c++;

Beg[i]->Key = c; i++;

}

**else** i++;

}

**else** cout << "Граф отсутствует!";

}

**void** Spisok::Print\_Color ()

**//Функция вывода раскраски графа, заданного**

**//модифицированными списками смежности Beg.**

{

**for** (**int** i=1;i<=XRY;i++)

**if** ( Beg[i] != **NULL** )

cout << " Color " << i << " - " << Beg[i]->Key << endl;

}

Текст этой программы можно взять [здесь](http://khpi-iip.mipk.kharkiv.edu/library/datastr/book_sod/kgsu/din116_3.zip).

Результат работы приложения изображен на рисунке 1:

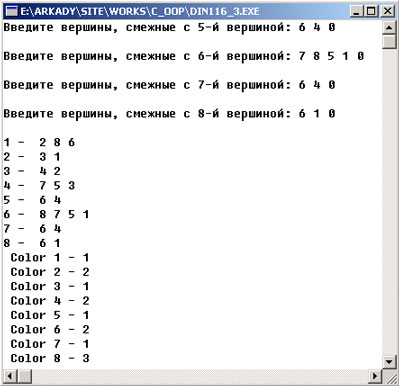


Рис.1. Результат работы приложения

Отметим, что для некоторых классов графов (например, полных **k**-дольных) последовательная раскраска является минимальной [2, с.238].

***В общем случае это не так!***

***Алгоритм прямого неявного перебора***

Приведем теперь ***алгоритм прямого неявного перебора*** для определения хроматического числа графа, использующий дерево поиска [3, с.89-90].

Предположим, что множество вершин каким-то образом упорядочено и **vi** - **i**-я вершина этого множества.

Вначале построим ***первоначальную допустимую раскраску*** по следующему алгоритму:

1) окрасить v1 в цвет 1;

2) каждую из оставшихся вершин окрашивать последовательно в соответствии с заданным упорядочением: вершина **vi** окрашивается в цвет с наименьшим возможным "номером" (т.е. выбираемый цвет должен быть первым в данном упорядочении цветом, не использованным при окраске какой-либо вершины, смежной с **vi**).

Опишем ***последующие шаги алгоритма*** согласно монографии [3, с.89].

Пусть **q** - число цветов, требуемое для вышеупомянутой раскраски. Если существует раскраска, использующая только **q-1** цветов, то все вершины, окрашенные в цвет **q**, должны быть перекрашены в цвет **j<q**. Пусть **vi\*** - первая вершина при заданном упорядочении, которая была окрашена в цвет **q**. Поскольку она была так окрашена потому, что не могла быть окрашена в цвет **j<q**, то ее можно перекрасить в цвет j

Итак, шаг возвращения из вершины **vi\*** можно осуществить следующим образом. Из смежных с **vi\*** вершин в множестве **{x1, x2, ..., xi\*-1}** найти последнюю (при заданном упорядочении), т.е. вершину с наибольшим индексом. Пусть это будет вершина **xk**. Если **xk** окрашена в цвет**jk**, то **xk** перекрашивается в другой допустимый цвет с наименьшим возможным "номером" **jk'**, таким, что **jk' >= jk+1**.

Если **jk'<q**, то надо продолжать последовательно перекрашивать все вершины с **xk+1** до **xn**, применяя указанное выше правило 2 и помня о том, что цвет **q** использовать нельзя.

Если такая процедура осуществима, то будет найдена новая лучшая раскраска, использующая меньше, чем **q** цветов. В противном случае, т.е. если встретится вершина, "требующая" цвет **q**, то можно снова сделать шаг возвращения - из такой вершины.

Если **jk'=q** или нет другого допустимого цвета **jk'**, то можно сразу же делать шаг возвращения из вершины **xk**.

Алгоритм заканчивает работу, когда на шаге возвращения достигается вершина **x1**.

Хотя описанная процедура неявного перебора является примитивным древовидным поиском, тем не менее она ничуть не хуже других известных методов раскраски графов [3, с.90].

***Замечание [1, с.70].*** *Мы не привели пока ни одного интересного примера, в котором хроматическое число выступает в роли объекта, практическая важность изучения которого несомненна. Попытаемся восполнить этот пробел.*

*Задача об обслуживании авиалиний.*

*Имеется некоторый город* ***M****, который связан маршрутами с городами* ***A1, A2, ..., Am****. Пусть, согласно расписанию, маршрут* ***MAiM*** *обслуживается в интервале времени* ***[ai, bi]****. Другими словами,* ***ai*** *- это тот момент, начиная с которого самолет связан с маршрутом* ***MAiM****, а* ***bi*** *- тот момент, когда эта связь прекращается. Таким образом, задано* ***m*** *временных интервалов* ***[a1, b1], ..., [am, bm]****. Требуется указать, каково минимальное число самолетов, достаточное для обслуживания всех рейсов?*

*Соответствующему расписанию сопоставим граф* ***G****, вершинами которого будут заданные временные интервалы, а две вершины будут смежны тогда и только тогда, когда соответствующие интервалы имеют непустое пересечение. Этот граф и служит моделью для нашей задачи.*

*Утверждение.*

*Искомое число самолетов равно хроматическому числу графа* ***G****.*

*Действительно, все рейсы, имеющие одинаковый цвет, могут быть обслужены одним самолетом. С другой стороны, если имеется некоторое число самолетов для обслуживания всех рейсов, то, окрасив одним цветом все рейсы, обслуживаемые одним самолетом, мы получим некоторую правильную раскраску графа* ***G****. Эти соображения и обосновывают сформулированное утверждение.*

(1) Компьютер и задачи выбора. - М.: Наука, 1989. - 208 с.

(2) Лекции по теории графов / Емеличев В.А., Мельников О.И., Сарванов В.И., Тышкевич Р.И. - М.: Наука, 1990. - 384 с.

(3) Кристофидес Н. Теория графов. Алгоритмический подход. - М.: Мир, 1978. - 432 с.

На следующем шаге мы рассмотрим ***изоморфизм графов***.

## Изоморфизм

На этом шаге мы рассмотрим ***алгоритмы, определяющие изоморфизм графов***.

Напомним сначала Вам несколько определений.

***Определение 1.***

Пусть **G** и **H** - графы, **f** - взаимно однозначное отображение множества **V(G)** на множество **V(H)** и **g** - взаимно однозначное отображение **E(G)** на **E(H)**. Обозначим **Q** упорядоченную пару **(f,g)**. Будем говорить, что **Q** есть***изоморфное отображение (изоморфизм)*** графа **G** на граф **H**, если выполняется следующее условие: вершина **x**инцидентна ребру **A** в графе **G** тогда и только тогда, когда вершина **fx** инцидентна ребру **gA** в графе **H**.

Если такой изоморфизм **Q** существует, то будем говорить, что графы **G** и **H** ***изоморфны***. Ясно, что в этом случае

|V(G)| = |V(H)| и |E(G)| = |E(H)|.

Мы можем рассматривать **Q** как операцию, преобразующую граф **G** в граф **H**, и в соответствии с этим писать **QG=H**. Удобно также писать **Qv=fv** и **QA=gA** (для каждой вершины v и каждого ребра **A** графа **G**).

***Автоморфизмом*** графа **G** называется изоморфизм графа **G** на себя.

Всякий граф **G** имеет ***тождественный*** (или ***тривиальный***) автоморфизм **I**, такой, что **Ix=x** для каждого ребра **x** и каждой вершины **x** из **G**.

Можно показать [1, с.21], что отношение изоморфизма между графами является ***отношением эквивалентности***, т.е. оно симметрично, транзитивно и рефлексивно. Следовательно, оно разбивает класс всех графов на непустые и попарно непересекающиеся подклассы, называемые ***классами изоморфизма*** или ***классами изоморфных графов***. Два произвольных графа принадлежат одному и тому же классу изоморфизма тогда и только тогда, когда они изоморфны друг другу.

Изоморфные графы естественно отождествлять, т.е. считать совпадающими (их можно изобразить одним рисунком). Они могли бы различаться конкретной природой своих элементов, но именно это игнорируется при введении понятия "граф".

Математическая теория графов интересуется такими свойствами графов, которые инвариантны относительно изоморфизма (например, числом вершин в графе, числом петель или числом вершин данной валентности и т.д.). Для специалиста по теории графов естественно отождествлять изоморфные графы. Поэтому обычно речь идет о классах изоморфизма (называемых также ***абстрактными графами***).

Вопрос о том, изоморфны ли два данных графа, в общем случае оказывается сложным (см., например, [2]).

Для изоморфизма двух n-вершинных графов само определение этого отношения дает теоретически безукоризненный способ проверки: надо просмотреть все **n!** взаимно однозначных соответствий между множествами вершин и установить, совмещаются ли полностью ребра графа хотя бы при одном соответствии. Однако даже весьма грубая оценка показывает, что такое решение задачи "в лоб" практически непригодно: уже при **n**=20 перебор всех **n!**вариантов потребовал бы около 40 лет машинного времени.

Подобная ситуация, естественно, толкнула многих математиков на классический путь: попытаться найти такой ***инвариант*** (число или систему чисел), который бы, с одной стороны, легко вычислялся по заданному графу (и по возможности имел наглядный смысл), а с другой - обладал ***свойством полноты***, т.е. определял граф однозначно с точностью до изоморфизма.

Вначале естественно поставить вопрос: какие характеристики графов инвариантны относительно изоморфизма [3,с.21]?

Примеры таких инвариантов графа **G** известны: это число вершин **n(G)**, число ребер **m(G)** и вектор степеней **s(G)=(s1, s2, ...,sn)**, который, в частности, дает числовые инварианты

min si и max si

i принадлежит {1,2,...,n} i принадлежит {1,2,...,n}

Второй из них часто называется ***степенью графа***.

***Определение 2.***

Пусть **f** - функция, относящая каждому графу **G** некоторый элемент **f(G)** из множества **M** произвольной природы. Эту функцию мы будем называть ***инвариантом***, если на изоморфных графах ее значения совпадают, т.е. для любых **G** и **G'** из изоморфности графов **G** и **G'** следует **f(G)=f(G')**.

Введем несколько наиболее важных инвариантов графа [3, с.21-25]:

* ***плотность* f(G)** - число вершин клики графа **G**. Инвариантность этой характеристики следует из того, что при изоморфном соответствии двух графов каждому подмножеству вершин одного графа, порождающему клику, соответствует в другом графе подмножество с тем же числом вершин и тоже порождающее клику;
* ***неплотность* e(G)** - это наибольшее количество попарно несмежных вершин графа;
* ***xроматическое число* g(G)**;
* ***число компонент связности* K(G)**;
* ***число Хадвигера* h(G)**.

Заметим, что ***матрица смежностей не является инвариантом графа***: при переходе от одной нумерации его вершин к другой она претерпевает ***перестановку рядов***, состоящую из некоторой перестановки строк и точно такой же перестановки столбцов. Любая функция от элементов **aij** матрицы, не меняющаяся ни при каких перестановках рядов матрицы смежности, является ***инвариантом графа* G**. К числу таких функций относятся сумма всех элементов, неупорядоченный набор сумм элементов каждой строки или сумм элементов каждого столбца, определитель матрицы, ее характеристический многочлен и корни последнего и др.

***Определение 3 [3, с.41-42].***

Инвариант **f** называется ***полным***, если для любых графов **G** и **G'** из равенства **f(G)=f(G')** следует изоморфизм графов **G** и **G'**.

Из рассмотренных инвариантов, которые отнесены к "легко вычислимым", даже наиболее "богатый" - вектор степеней **s(G)** - не является полным. В процессе развития теории графов не было нехватки в гипотезах полноты или иного "трудно вычислимого" инварианта, но все эти гипотезы рано или поздно опровергались конкретными примерами.

***Определение 4.***

Степенью **s** вершины **v** графа **(V,E)** называется число его вершин, смежных с **v**, или, что то же, число ребер, инцидентных этой вершине.

Ясно, что при всяком изоморфизме графов **L** и **L'** соответствующие друг другу вершины должны иметь ***одинаковую степень***.

***Определение 5 [3,с.17].***

Пусть **G=(V,E)** - **n**-вершинный граф, а **s1, s2, ...,sn** - степени его вершин, выписанные в порядке неубывания: **s1<= s2 <= ... <= sn**.

Упорядоченную систему **(s1, s2, ...,sn)** будем называть ***вектором степеней графа* G** и кратко обозначать **s(G)**.

Вместо самого вектора степеней часто пользуются его ***обращением*** **t(G)=(t1, t2, ...,tn)**, где **ti=sn-i (i=1,2,...,n)** - те же степени вершин, но расположенные в порядке невозрастания: **t1 >= t2 >= ... >= tn**.

Из сказанного выше следует, что для изоморфизма графов **G** и **G'** ***необходимо*** совпадение векторов их степеней: **s(G)=s(G')**.

Однако ***достаточным это условие не является***: на рисунке 1 мы видим две пары неизоморфных графов с одинаковыми векторами: **s = (1,2,2,2,2,3)**:

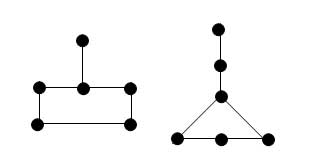


Рис.1. Пара неизоморфных графов

Не будучи идеальным средством распознавания изоморфизма, вектор степеней может тем не менее во многих случаях оказать существенную помощь.

***Во-первых***, если **s(G)** не равно **s(G')**, то отсюда сразу следует неизоморфность графов **G** и **G'**.

***Во-вторых***, если **s(G)=s(G')**, то для проверки графов **G** и **G'** на изоморфизм требуется перебор не всех **n!**соответствий между вершинами, а лишь тех, при которых сопоставляются вершины одинаковой степени. Так в примере надо перебрать лишь **4!=24** соответствия вместо 720 [3,с.17].

Пример. ***Распознавание изоморфизма неориентированных графов с помощью анализа вектора степеней***. Результатом работы программы являются все взаимно-однозначные отоб 1ражения множества вершин первого графа на множество вершин второго графа, определяющие изоморфизм.

В программе существенным образом использован модификация алгоритма ***Джонсона*** и ***Троттера*** генерирования всех перестановок с минимальным числом транспозиций соседних элементов.

**#include** <iostream.h>

**#define** TRUE 1

**#define** FALSE 0

**#define** M 10 **//Максимальное количество вершин в графе.**

**typedef** **int** Boolean;

**typedef** **struct** L \*Lref; **// Тип: указатель на заголовочный узел.**

**typedef** **struct** T \*Tref; **// Тип: указатель на дуговой узел.**

**//Описание типа заголовочного узла.**

**typedef** **struct** L

{

**int** Key; **//Имя заголовочного узла.**

**int** Count; **//Количество предшественников.**

Tref Trail; **//Указатель на список смежности.**

Lref Next; **//Указатель на следующий узел в списке заголовочных узлов.**

};

**//Описание типа дугового узла.**

**typedef** **struct** T

{

Lref Id;

Tref Next;

};

**class** Spisok

{

**private**:

Lref Head; **//Указатель на голову списка заголовочных узлов.**

Lref Tail; **//Указатель на фиктивный элемент**

**// в конце списка заголовочных узлов.**

**int** Fun[M+1]; **//Массив вершин графа.**

**int** NewFun[M+1]; **//Массив перестановок элементов массива Fun.**

**void** SearchGraph (int, Lref \*);

Lref Search(**int**);

Boolean Find\_Arc (int, int);

Boolean Isomorph (Spisok, **int**);

**public**:

Spisok() {**//Инициализация списка заголовочных узлов.**

Head = Tail = **new** (L); }

Lref GetHead() { **return** Head; }

Lref GetTail() { **return** Tail; }

**void** MakeGraph ();

**void** PrintGraph ();

**void** Sorting ();

**friend void** Reshenie(Spisok,Spisok);

};

**void** main ()

{

Spisok A,B;

**//Построение графов и вывод их структур смежности.**

A.MakeGraph ();

A.PrintGraph (); cout<<endl;

B.MakeGraph ();

B.PrintGraph (); cout<<endl;

A.Sorting ();

A.PrintGraph (); cout<<endl;

B.Sorting ();

B.PrintGraph (); cout<<endl;

**//Решение.**

Reshenie(A,B);

}

**void** Reshenie(Spisok A, Spisok B)

**//Решение задачи.**

{

Lref Ukaz; **//Рабочий указатель для перемещения**

**// по списку заголовочных звеньев.**

**int** P[M+1]; **//Массив для перестановок.**

**int** Ves[M+1]; **//Массив "весов" элементов массива P**

**//(в нем хранятся степени вершин).**

**int** Ves1[M+1]; **//Вначале - копия массива Ves,**

**//затем - рабочий массив.**

Boolean Fl; **//Флаг совпадения массивов Ves и Ves1.**

Boolean PR[M+1];

**int** C[M+1];

**int** x,k,t;

**//Определение количества вершин и функций.**

**int** N=0;

Ukaz = A.GetHead();

**while** (Ukaz!=A.GetTail())

{ N++;

A.Fun[N] = A.NewFun[N] = Ukaz->Key; Ves[N] = Ukaz->Count;

Ves1[N] = Ves[N]; Ukaz = Ukaz->Next;

}

N=0;

Ukaz = B.GetHead();

**while** (Ukaz!=B.GetTail())

{ N++;

B.Fun[N] = Ukaz->Key; Ukaz = Ukaz->Next;

}

**for** (**int** i=1;i<=N;i++) cout << Ves[i] << " "; cout << endl;

**for** (i=1;i<=N;i++) cout << A.Fun[i] << " "; cout << endl;

**for** (i=1;i<=N;i++) cout << B.Fun[i] << " "; cout << endl;

**//Инициализация.**

**for** (i=1;i<=N;i++)

{ P[i] = i; C[i] = 1; PR[i] = TRUE; }

**//Нахождение перестановок.**

C[N] = 0;

cout << "Отображение вершин, определяющее изоморфизм: \n";

**if** ( A.Isomorph (B,N) )

{

**for**(**int** j=1;j<=N;j++) A.NewFun[j] = A.Fun[P[j]];

**for**(j=1;j<=N;j++) cout << A.NewFun[j] << " ";

cout << endl;

**for**(j=1;j<=N;j++) cout << B.Fun[j] << " ";

cout << endl;

}

cout << " --------------------------- \n";

i = 1;

**while** ( i<N )

{

i = 1; x = 0;

**while** ( C[i]==N-i+1 )

{

PR[i] = (!PR[i]); C[i] = 1;

**if** ( PR[i]) x++;

i++;

}

**if** ( i<N )

**//Выполнение транспозиции.**

{

**if** ( PR[i] ) k = C[i] + x;

**else** k = N - i + 1 - C[i] + x;

t = P[k]; P[k] = P[k+1]; P[k+1] = t;

t = Ves1[k]; Ves1[k] = Ves1[k+1];

Ves1[k+1] = t; C[i] += 1;

**//Отбор нужной перестановки.**

Fl = TRUE;

**for** (**int** j=1;j<=N;j++) Fl = (Fl && (Ves[j]==Ves1[j]));

**if** ( Fl )

{

**for** (j=1;j<=N;j++) A.NewFun[j] = A.Fun[P[j]];

**if** (A.Isomorph (B,N))

{

**for** (j=1;j<=N;j++) cout << A.NewFun[j] << " ";

cout << endl;

**for** (j=1;j<=N;j++) cout << B.Fun[j] << " ";

cout << endl;

cout << " --------------------------- \n";

}

}

}

}

}

Lref Spisok::Search (**int** w)

**//Функция возвращает указатель на заголовочный узел**

**//ключом w. Если узел отсутствует, то функция возвращает NULL .**

{

Lref h = Head;

(\*Tail).Key = w; **//Поиск "с барьером".**

**while** ((\*h).Key!=w) h = (\*h).Next;

**if** (h==Tail) **//В списке заголовочных узлов нет узла с ключом w.**

h = **NULL**;

**return** h;

}

**void** Spisok::SearchGraph (**int** w, Lref \*h)

**//Функция возвращает указатель на заголовочный узел**

**//с ключом w в графе, заданном структурой Вирта с указателем Head.**

{

\*h = Head; (\*Tail).Key = w;

**while** ((\*\*h).Key!=w) \*h = (\*\*h).Next;

**if** (\*h==Tail)

**//В списке заголовочных узлов нет узла с ключом w.**

**//Поместим его в конец списка Head.**

{ Tail = **new** (L); (\*\*h).Count = 0;

(\*\*h).Trail = **NULL**; (\*\*h).Next = Tail; }

}

**void** Spisok::MakeGraph ()

**//Функция возвращает указатель Head на структуру**

**//Вирта, соответствующую ориентированному графу.**

{

**int** x,y;

Lref p,q; **//Рабочие указатели.**

Tref t,r; **//Рабочие указатели.**

Boolean Res; **//Флаг наличия дуги.**

cout<<"Вводите начальную вершину дуги: ";

cin>>x;

**while** (x!=0)

{

cout<<"Вводите конечную вершину дуги: "; cin>>y;

**//Определим, существует ли в графе дуга (x,y)?**

SearchGraph (x, &p); SearchGraph (y,&q);

r = (\*p).Trail; Res = FALSE;

**while** ((r!=**NULL**)&&(!Res))

**if** ((\*r).Id==q) Res = TRUE;

**else** r = (\*r).Next;

**if** (!Res) **//Если дуга отсутствует, то поместим её в граф.**

{ t = **new** (T); (\*t).Id = q;

(\*t).Next = (\*p).Trail; (\*p).Trail = t; (\*q).Count++; }

cout<<"Вводите начальную вершину дуги: "; cin>>x;

}

}

**void** Spisok::PrintGraph ()

**//Вывод структуры Вирта, заданной указателем**

**//Head и соответствующей ориентированному графу.**

{

Lref p; **//Рабочий указатель.**

Tref q; **//Рабочий указатель.**

p = Head;

**while** (p!=Tail)

{

cout<< "(" << (\*p).Count << ")" << (\*p).Key << "("; q = (\*p).Trail;

**while** (q!=**NULL**)

{ cout<<(\*(\*q).Id).Key; q = (\*q).Next; }

cout<<")"; p = (\*p).Next; cout<<" ";

}

}

Boolean Spisok::Find\_Arc (**int** x, **int** y)

**//Функция возвращает TRUE, если в графе, представленном**

**//структурой Вирта (Head,Tail), существует дуга (x,y).**

{

Lref p,q; **//Рабочие указатели.**

Tref r; **//Рабочие указатели.**

Boolean Res; **//Флаг наличия дуги.**

**//Определим, существует ли в графе дуга (x,y)?**

p = Search (x);

q = Search (y);

r = p->Trail; Res = FALSE;

**while** ( r != **NULL** && !Res )

**if** (r->Id==q) Res = TRUE;

**else** r = r->Next;

**return** Res;

}

**void** Spisok::Sorting ()

**//Сортировка узлов в заголовочном списке**

**//по полю Count в порядке убывания.**

{

Lref UkZv\_1,UkZv\_2;

Lref UkZv\_3;

Tref UkZv\_4;

**int** A, B;

Tref D;

UkZv\_1 = Head;

**while** ( UkZv\_1 != Tail )

{

UkZv\_2 = UkZv\_1->Next;

**while** ( UkZv\_2!=Tail )

{

**if** ( UkZv\_1->Count < UkZv\_2->Count )

{

A = UkZv\_1->Key;

B = UkZv\_1->Count;

D = UkZv\_1->Trail;

UkZv\_1->Key = UkZv\_2->Key;

UkZv\_1->Count = UkZv\_2->Count;

UkZv\_1->Trail = UkZv\_2->Trail;

UkZv\_2->Key = A;

UkZv\_2->Count = B;

UkZv\_2->Trail = D;

UkZv\_3 = Head;

**while** ( UkZv\_3!=Tail )

{

UkZv\_4 = UkZv\_3->Trail;

**while** ( UkZv\_4 != **NULL** )

{

**if** ( UkZv\_4->Id==UkZv\_1 ) UkZv\_4->Id = UkZv\_2;

**else**

**if** ( UkZv\_4->Id==UkZv\_2 ) UkZv\_4->Id=UkZv\_1;

UkZv\_4 = UkZv\_4->Next;

}

UkZv\_3 = UkZv\_3->Next;

}

}

UkZv\_2 = UkZv\_2->Next;

}

UkZv\_1 = UkZv\_1->Next;

}

}

Boolean Spisok::Isomorph (Spisok B, **int** N)

**//Функция, возвращающая TRUE, если графы, представленные**

**//структурами Вирта (A.Head,A.Tail) и (B.Head,B.Tail), изоморфны.**

**//Отображение вершин при изоморфизме задано векторами**

**//NewFun и Fun.**

{

Boolean Flag = TRUE;

**int** k1,k2;

**for** (**int** i=1;i<=N;i++)

{

k1 = 1;

**while** ( i != this->NewFun[k1] ) k1++;

**for** (**int** j=1;j<=N;j++)

{

k2 = 1;

**while** ( j != this->NewFun[k2] ) k2++;

**if** ( i>j && this->Find\_Arc(i,j) )

Flag = (Flag && B.Find\_Arc (B.Fun[k1],B.Fun[k2]));

}

}

**return** Flag;

}

Текст этой программы можно взять [здесь](http://khpi-iip.mipk.kharkiv.edu/library/datastr/book_sod/kgsu/din117_1.zip).

Для графов, изображенных на рисунке 2,

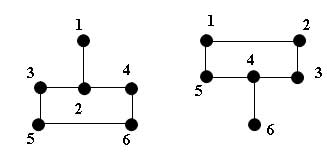


Рис.2. Пример графов

результат работы приложения будет следующим:

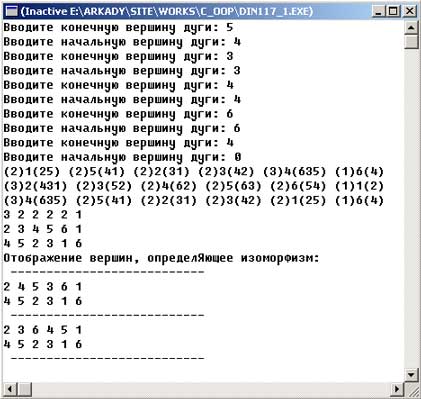


Рис.3. Результат работы приложения

Нетривиальный тестовый пример Вы можете обнаружить в монографии [4, с.207].

Однако есть случаи, когда при выяснении изоморфизма графов их векторы степеней практически бесполезны: речь идет об ***однородных*** графах, в которых степень всех вершин одна и та же.

Например, взгляните на граф, степени всех вершин которого равны 3:

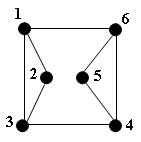


Рис.4. пример однородного графа

Противоположный случай представляют графы, определяемые однозначно с точностью до изоморфизма своим вектором степеней (или, что равносильно, его обращением) и называемые ***униграфами***.

***Алгоритм, распознавания изоморфизма графов, описанный в [5]***

Этот подход используется при построении алгоритма установления изоморфизма ориентированных графов в монографии [5,с.398].

Предположим, что нам даны два ориентированных графа **GX=(VX,EX)** и **GY=(VY, EY)** и требуется выяснить, изоморфны ли они. Мы полагаем, что **VX=VY={1,2,...,n}**, ибо если **|VX|** не равно **|VY|**, то ориентированные графы не могут быть изоморфными.

Пусть один из ориентированных графов, скажем **GX**, выбран в качестве эталона. Пусть **GX(k)** - подграф графа **GX**, индуцируемый вершинами **{1,2,...,k}**, **0<=k<=n**. Ясно, что **GX(0)** - пустой подграф и **GX(1)** - подграф, состоящий из единственной вершины 1 и не содержащий ребер.

При определении того, изоморфны ли графы **GX** и **GY**, используем ***технику поиска с возвращением***.

Очевидно, **GX(0)** изоморфен пустому подграфу **GY**. Предположим, что на некотором шаге найден подграф **GY**, состоящий из вершин **S**, принадлежащих VY, который изоморфен **GX(k)**.

Попытаемся продолжить изоморфизм на **GX(k+1)**, выбирая вершину **v**, принадлежащую **VY-S**, соответствующую **k+1** из **VX**.

Если такая вершина **v** найдена, то зафиксируем соответствие **fk+1<-v** и попытаемся продолжить изоморфизм на **GX(k+2)**.

Если такой вершины не существует, то возвращаемся в **GX(k-1)** и пытаемся выбрать другую вершину, соответствующую **k** из **VX**.

Этот процесс продолжается до тех пор, пока не будет найден изоморфизм между **GX(n)=GX** и **GY**, в противном случае возвращаемся к **GX(0)**, заключив, что графы **GX** и **GY** неизоморфны.

***С.Гудман*** и ***С.Хидетниеми*** [6,с.62] предложили называть ***эвристическим*** алгоритм, который обладает следующими двумя свойствами:

1. Он находит, как правило, хорошие, хотя и не обязательно оптимальные решения.
2. Его легче и быстрее реализовать, чем любой из известных точных алгоритмов (т.е. алгоритмов, гарантирующих оптимальное решение).

Эвристики для решения задач изоморфизма обычно состоят в попытках показать, что рассматриваемые графы не изоморфны. Для этого составляется ***список различных инвариантов*** в порядке, определяемом сложностью вычисления инварианта. Затем последовательно сравниваются значения параметров данных графов. Как только обнаруживаются два различных значения одного и того же параметра, приходят к заключению, что данные графы ***неизоморфны***.

Рассмотрим пример достаточно сложной эвристики, работающей с матрицей смежности **A(G)** [6, с.63-64].

***Алгоритм сравнения порядков смежности***

Вычисляется **A2(Gi)** для **i=1, 2**. Затем переставляются строки и столбцы **A2(G1)** и **A2(G2)** так, чтобы элементы на главной диагонали оказались в нисходящем порядке.

Если **G1** и **G2** изоморфны и все диагональные элементы различны, то при этой перестановке должны получиться идентичные матрицы.

Если нет, то данные два графа не могут быть изоморфными.

Если матрицы идентичны, то можно продолжить проверку с **A3(Gi)**, **A4(Gi), ..., Ak(Gi)** для **i=1, 2**. Значение **k**определяется имеющимся бюджетом машинных ресурсов. Если все из проверенных матриц совпадают, то весьма правдоподобно, но не обязательно истинно, что **G1** и **G2** изоморфны.

***Замечания.***

1. *Графы изоморфны тогда и только тогда, когда их* ***матрицы смежностей*** *получаются друг из друга* ***одинаковыми*** *перестановками строк и столбцов [7,с.28].*

*Графы (ориентированные графы) изоморфны тогда и только тогда, когда их* ***матрицы инцидентности****получаются друг из друга* ***произвольными*** *перестановками строк и столбцов [7, с.28].*

1. *Существует немало глубоких и трудных проблем, связанных с изоморфизмом графов, и наиболее известная из них -* ***проблема восстановления****.*

*Пусть вершины графа* ***G*** *занумерованы следующим образом:* ***v1, v2, ...,vk****. Через* ***Gj*** *обозначим граф, получающийся из* ***G*** *после удаления вершины* ***vj*** *и инцидентных ей ребер. Все* ***k*** *подграфов* ***Gj*** *0 графа* ***G*** *будем называть* ***примарными*** *[1, с.154].*

***C1 (гипотеза о восстановлении).***

*Если заданы классы изоморфизма всех* ***k*** *примарных подграфов графа* ***G****, то при* ***k>=3*** *класс изоморфизма графа* ***G*** *определяется однозначно.*

*К моменту написания книги У.Татта [1] гипотеза о восстановлении оставалась недоказанной.*

*Ограничение на* ***k (k>=3)*** *обосновывается легко.*

*Если* ***k=1****, то единственным примарным подграфом графа* ***G*** *будет пустой граф, который не несет никакой информации о числе петель в графе* ***G****. Если* ***k=2****, то в каждом из двух примарных подграфов графа* ***G****содержится только по одной вершине, а потому нам ничего не известно о числе ребер в графе* ***G****. Таким образом, действительно нужно предположить, что* ***|V(G)|>=3****.*

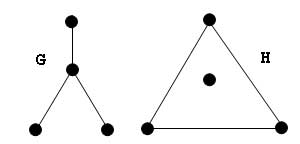
*Если некоторое свойство или характеристику графа* ***G*** *можно выявить, рассматривая классы изоморфизма всех его примарных подграфов* ***Gj****, то оно (она) называется* ***восстанавливаемым (восстанавливаемой)****, а графы, обладающие этим свойством или характеристикой, называются* ***распознаваемыми****. Примером восстанавливаемой характеристики является число вершин в графе. Это число на единицу больше числа вершин в каждом примарном подграфе данного графа и совпадает с числом заданных классов изоморфизма.*

*Пусть* ***G*** *- граф с занумерованными ребрами* ***A1, A2, ...,Ak****. Через* ***Gj*** *обозначим подграф графа* ***G****, получающийся после удаления из* ***G*** *ребра* ***Aj****.*

***C2 (гипотеза о реберном восстановлении).***

*Если заданы классы изоморфизма всех* ***k*** *подграфов* ***Gj****, то при* ***k>=4*** *класс изоморфизма графа* ***G****определяется однозначно.*

*Причина, по которой введено ограничение на* ***k (k>=4)*** *ясна из следующего рисунка [1, с.164], на котором изображены два графа* ***G*** *и* ***H****, такие, что, удаляя из них произвольное ребро, мы получаем, с точностью до изоморфизма, один и тот же подграф.*

**

*Рис.5. Пример графов*

1. *[7, с.14]. В некоторых ситуациях все же приходится различать изоморфные графы, и тогда полезно понятие "помеченный граф". Граф порядка* ***n*** *называется* ***помеченным****, если его вершинам присвоены некоторые метки, например, номера* ***1, 2, ..., n****.*

*Отождествив каждую из вершин графа с ее номером (и, следовательно, множество вершин - с множеством чисел* ***{1, 2, ..., n}****), определим* ***равенство помеченных графов******G*** *и* ***H*** *одного и того же порядка:* ***G=H*** *тогда, когда множество ребер графа* ***G*** *совпадает с множеством ребер графа* ***H****.*

(1) Татт У. Теория графов. - М.: Мир, 1988. - 424 с.

(2) Земляченко В.Н., Корнеенко Н.М., Тышкевич Р.И. Проблема изоморфизма графов // Теория сложности вычислений, I. Записки научных семинаров ЛОМИ. - 1982. - Т.118. - С.83-158.

(3) Зыков А.А. Основы теории графов. М.: Наука, 1987. - 384 с.

(4) Белов В.В., Воробьев Е.М., Шаталов В.Е. Теория графов. - М.: Высш.шк., 1976. - 392 с.

(5) Рейнгольд Э., Hивергельт Ю., Део H. Комбинаторные алгоритмы. Теория и практика. - М.: Мир, 1980. - 476 с.

(6) Гудман С., Хидетниеми С. Введение в разработку и анализ алгоритмов. - М.: Мир, 1981. - 368 с.

(7) Лекции по теории графов / Емеличев В.А., Мельников О.И., Сарванов В.И., Тышкевич Р.И. - М.: Наука, 1990. - 384 с.

Со следующего шага мы начнем краткий обзор ***сетей***.

1. Эволюция и революция

## Второй уровень

### Третий уровень

Четвертый уровень

#### Пятый уровень

Основной текст.

* Список

1. Первый элемент списка.
2. Второй элемент списка

Рис. 1

p p p p p p p p p p p p p p p p p p