Лекция 7. Графы: способы их хранения и обхода (в ширину и в глубину). Проверка графа на двудольность, поиск циклов и топологическая сортировка графа

Понятие графа. Специальные графы. Способы хранения графов в виде матрицы смежности и списка смежности. Способы обхода графа: обход в ширину и обход в глубину. Компоненты связности графа. Ориентированные графы. Поиск циклов в ориентированном и неориентированном графе. Проверка графа на двудольность. Топологическая сортировка графа. Расстояния в графе.

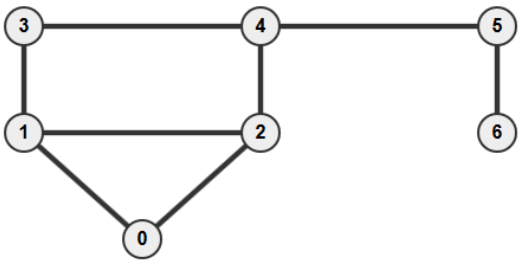
Общее представление

Основные понятие теории графов занимают важное место и в математических олимпиадах, и в олимпиадах по программированию. Почти в каждой олимпиаде по программированию будет встречаться задача на тему «Графы» и для того, чтобы ее решить необходимо не только хорошо знать известные алгоритмы на графах, но и быть математически осведомленным в теории графов. В этой лекции будут освещены самые необходимые понятия и алгоритмы, без которых невозможно обойтись. Кроме того, графы занимают важное место в прикладных исследованиях по различным разделам знаний: в геоинформатике, в химии, экономике и теории управления, в логистике, биоинформатике. Лишь уверенно освоив базовые алгоритмы, приведенные в этой лекции, можно будет приступать к дальнейшему изучению более сложных алгоритмов теории графов.

Основные определения графов. Способы хранения графов

Графом G называется совокупность множеств $\mathrm{G} = \big(\mathrm{V(G)}, \mathrm{E(G)}\big)$, где $\mathrm{V(G)}$ — непустое конечное множество элементов, называемых *вершинами графа*, а $\mathrm{E(G)}$ — множество пар элементов из $\mathrm{V(G)}$ (необязательно различных), называемых *ребрами графа*. $\mathrm{E(G)} = \{(v_i, v_j)\}$ — множество ребер графа G, состоящее из пар вершин (v_i, v_j). Ребро (v_i, v_j) соединяет вершины v_i и v_j. Две вершины, соединенные ребром, называют смежными вершинами. Количество ребер, исходящее из вершины называют степенью вершины *d(v)*. Если какие-то две вершины соединены более, чем одним ребром, то говорят, что граф содержит *кратные ребра*. Если ребро соединяет вершину саму с собой, то такое ребро называют *петлей*. *Простой граф* не содержит петель и кратных ребер. Граф называют *ориентированным*, если ребра графа имеют направление (в этом случае они называются *дугами*), в противном случае граф – *неориентированный*.

Матрица смежности A для данного графа G содержит элементы $a_{ij} = 1$, если две вершины v_i и v_j являются смежными и $a_{ij} = 0$, если вершины v_i и v_j смежными не являются. Матрица смежности простого графа симметрична. Пример графа G и матрица смежности A для данного графа приведены на рисунке.



Матрица смежностей

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

V={0, 1, 2, 3, 4, 5, 6} – множество вершин графа. |V|=7 – количество вершин.

E={ (0,1),(0,2),(1,2),(1,3),(2,4),(3,4),(4,5),(5,6)}. |E|=8 – количество ребер

Хранение матрицы смежности требует $O(|V|^2)$ памяти, что во многих случаях неэкономно. В ряде алгоритмов удобнее пользоваться не матрицей смежностей графа, а *списком смежности*. В списке смежности для каждой вершины указываются вершины, смежные с ней. В следующей таблице представлен список смежности вершин для графа G. Хранение списка смежности требует $O(|V| + |E|)$ памяти.

Если входные данные к задаче задают матрицу смежности графа, то по ней легко получить список смежности в виде двумерного массива g, где g[i] – вектор вершин, смежных с вершиной *i*.

Вершина	Смежные вершины		Преобразование матрицы смежности в список смежности
0	1, 2	d(0)=2	
1	0, 2, 3	d(1)=3	
2	0, 1, 4	d(2)=3	
3	1, 4	d(3)=2	
4	2, 3, 5	d(4)=3	
5	4, 6	d(5)=2	
6	5	d(6)=1	

```
vector<vector<int>> g(n);
   for (int i=0; i<n;++i)
     for (int j=0;j<n;++j){
          cin>>a;
          if (a) g[i].push_back(j);
     }
```

Если входные данные к задаче представляют собой список ребер, то по данному списку легко получаем и матрицу смежности, и список смежности. Ниже приведен пример для графа G входных данных в виде списка ребер: на вход программы поступают числа *n* – количество вершин в графе и *m* (1<=*m*<=*n(n-1)*/2) – количество ребер. Затем следует *m* пар чисел – ребра графа.

```
7  8
0  1
0  2
1  2
1  3
2  4
3  4
4  5
5  6
```

Входные данные: n=7, m=8. 8 ребер заданы парой смежных вершин. Считываем список ребер и формируем массив списка смежности g.

```
cin>>n>>m;
vector<vector<int>> g(n);
for (int i=0; i<m;++i){
          cin>>v1>>v2;
          g[v1].push_back(v2);
          g[v2].push_back(v1);
}
```

Плотные графы, имеющие большое количество ребер выгоднее хранить при помощи матрицы смежности, а вот *разреженные графы*, имеющие небольшое количество ребер, оптимальнее хранить при помощи списка смежности.

Если граф является ориентированным, то матрица смежности в общем случае не будет являться симметричной. Для *взвешенных графов*, то есть графов, каждому ребру которых сопоставлено число, например, расстояние от одной вершины до другой, соответствует матрица весов. Матрица весов является обобщением матриц смежности и содержит в качестве элементов веса ребер графа.

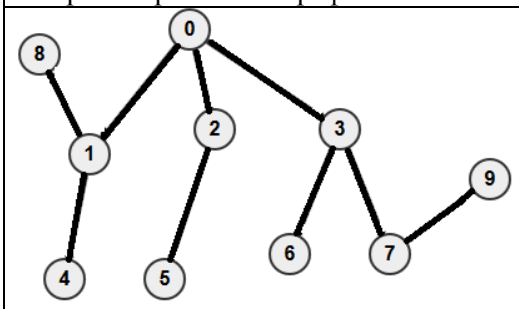
Обход графа в глубину

Алгоритм обхода графа глубину (depth-first search)

Обход графа в глубину DFS позволяет определить вершины, достижимые из данной вершины. При обходе графа используется массив used[], хранящий информацию о том, была ли посещена вершина: $used[i] = 1$, если вершина уже была посещена при обходе и $used[i] = 0$, если вершина еще не была посещена. В начале работы алгоритма все вершины являются непосещенными:

$used[i] = 0,\ \forall i = 1..n$. Функция DFS получает на вход очередную вершину и вызывает себя рекурсивно для всех еще непосещенных вершин, достижимых из данной. Из основного текста программы вызов dfs осуществляется от стартовой вершины – той, с которой мы начинаем обход.

Неориентированный граф G	Функция обхода графа в глубину DFS

	`void dfs (int v){`
	`  used[v]=1;`
	`  for (auto i=g[v].begin();i!=g[v].end();++i)`
	`  if(!used[*i])`
	`  dfs(*i);`
	`}`
	Вызов из основного текста программы
	`dfs (0)`

Программа обойдет вершины графа в следующем порядке: 0, 1, 4, 8, 2, 5, 3, 6, 7, 9.

Оценка сложности алгоритма включает в себя следующие операции:

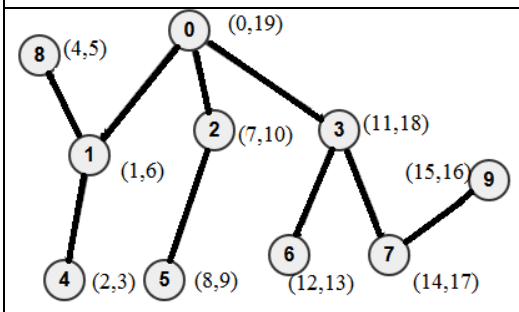
- просмотр всех |V| вершин, для каждой из которых *v* просматриваются ее соседи;
- просмотр всех соседей вершины *v*. При этом алгоритм проходит по ребру {v, u}. Причем, каждое такое ребро {v, u} просматривается дважды: для вершины *u* и для вершины *v*.

Итоговая сложность алгоритма dfs, таким образом $O(|V| + |E|)$.

Время начала и конца обработки вершины

Для ясного понимания алгоритма поиска в глубину рассмотрим время начала обработки и время конца обработки вершин. В счетчике time будет фиксироваться текущее время, каждый раз увеличиваясь на единицу. В вектора time_in[] будет записываться время начала обработки для вершин, а в вектора time_out[] – время конца обработки. После работы алгоритма dfs получим времена начала и конца обработки вершин графа G, указанные в виде пар чисел на графе.

Неориентированный граф G	Функция обхода графа в глубину DFS с фиксацией времени начала и конца обработки вершин

	`void dfs (int v){`
	`  used[v]=1;`
	`  time_in[v]=time++;`
	`  for (auto i=g[v].begin();i!=g[v].end();++i)`
	`  if(!used[*i])`
	`  dfs(*i);`
	`  time_out[v]=time++;`
	`}`

Заметим, что [time_in [v], time_out[v]] – это промежуток времени, в течение которого вершина v была на обработке. Аналогично, [time_in[u], time_out[u]] – это промежуток времени, в течение которого вершина u была на обработке. Указанные отрезки либо не пересекаются, либо содержится один в другом, так как если вершина u, например, начала обрабатываться после вершины v, то и завершение ее обработки будет раньше, чем у вершины v.

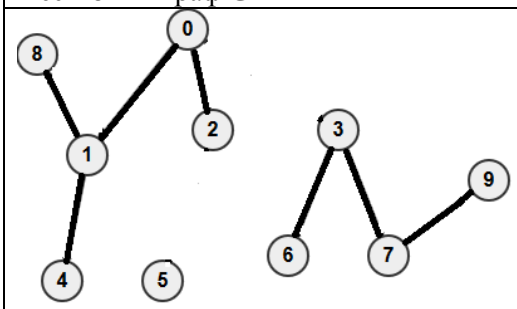
При помощи значений векторов time_in[] и time_out[] можно проверить, является ли одна вершина дерева предком другой. Ответ можно найти за O(1): вершина *i* является предком вершины *j* тогда и только тогда, когда time_in[i]<time_in[j] и time_out[i]>time_out[j].

Поиск компонент связанности графа

Путем в графе называется последовательность вершин $v_i \in V,\ i = 1..k$ таких, что две любые последовательные вершины соединены хотя бы одним ребром и все ребра различны. Число k вершин в пути называется *длиной пути*. Граф G называется *связным*, если две его любые вершины

соединены путем. Если граф не связен, то его можно разбить на непересекающиеся связные подмножества, называющиеся *компонентами связанности*. На рисунке представлен несвязный граф, имеющий три компоненты связности. Поиск глубину dfs будет обходить ту компоненту связности, из вершины которой, он был вызван. Поэтому при помощи dfs легко проверить, является ли граф связным, вызывая во внешнем цикле dfs от всех, еще непосещенных вершин. Можно также найти все компоненты связности и для каждой вершины вывести номер той компоненты связности, которой она принадлежит – номера компонент связности вершин будут храниться в векторе cc_num[].

Несвязный граф G	Вызов DFS из основного текста программы

 {0, 1, 2, 4, 8}, {5}, {3, 6, 7, 9} - компоненты связности.	`for (int i=0;i<n;++i){`
	`  if (!used[i]){`
	`  cc++;`
	`  dfs(i);`
	`  }`
	`}`
	В саму функцию DFS необходимо добавить заполнение вектора номеров компонент связности для вершин графа - `cc_num[v]=cc;`

Обход ориентированного графа в глубину

Рассмотрим особенности обхода графа в глубину для ориентированных графов. Входные данные для ориентированного графа в виде перечня ребер *u v*, при этом подразумевают, что направление ребра *(u,v)* - от вершины u к вершине v. При считывании данных добавляем только вершину v к списку смежности вершины u.

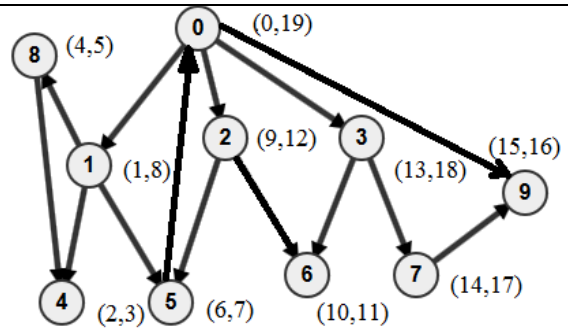
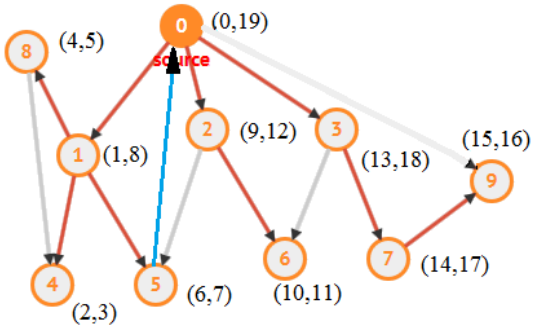
Алгоритм DFS как на неориентированном графе, так и на ориентированном графе построит дерево. *Дерево* – это связный граф без циклов. Корень дерева будет находиться в стартовой вершине (той, с которой начался обход в глубину). Ориентированный граф и соответствующее ему дерево приведены в таблице ниже. На графе справа можно выделить несколько типов ребер.

Древесные ребра. Красным цветом в дереве справа помечены те ребра *(u,v),* которые могут быть изображены в соответствии с временем начала и конца обработки вершин. Это так называемые древесные ребра. Для их вершин выполняется: time_in[u]<time_in[v]<time_out[v]<time_out[u].

Прямые ребра. Ведут от вершины к потомку, не являющемуся ребенком. Для них выполняется такое же условие, что и для древесных ребер *(u,v):* На рисунке справа ребро (0, 9) – прямое.

Обратные ребра. Ведут от вершины к ее предку. То есть, выполняется условие time_in[v]<time_in[u]<time_out[u]<time_out[v]. Синее ребро на рисунке – обратное.

Перекрестные ребра. Для таких ребер выполняется условие time_in[v]<time_out[v]< time_in[u]<time_out[u]. На рисунке справа можно найти перекрестные ребра, например (8,4).

Ориентированный граф G	Дерево, построенное алгоритмом DFS по ориентированному графу G
	

В случае *неориентированного* графа алгоритм в глубину также будет строить дерево на графе, а все ребра после алгоритма, примененного к неориентированному графу, будут двух типов: древесные и обратные.

Поиск циклов в графе

Циклом в графе G называется путь, ведущий из вершины *v* в саму себя. Граф называют *ациклличным*, если в нем нет циклов. Рассмотрим граф G без петель и кратных ребер, и проверим его на ацикличность. Функция dfs обнаружит цикл в том случае, если пытается пройти в вершину, которая находится в данный момент времени в обработке. Вершины, находящиеся в обработке, алгоритм будет красить в серый цвет, вершины, не находящиеся в обработки предварительно покрашены в белый цвет, а уже обработанные вершины алгоритм будет красить в черный цвет. Для вывода всех вершин цикла удобно использовать массив предков.

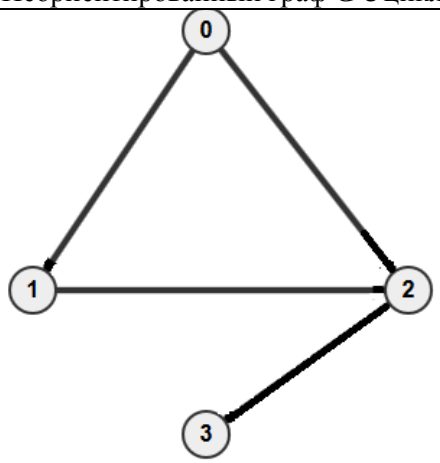
Поиск циклов в неориентированном графе

Для неориентированных графов нужно предусмотреть ситуации, когда dfs из вершины идет в ее предка – такие ситуации не являются признаком цикла. Для хранения массива предков будем использовать массив p[].

Неориентированный граф G. Поиск цикла

```
void dfs (int v) {
  used[v] = 1;
  for (size_t i=0; i<g[v].size(); ++i) {
     int to = g[v][i];
     if (used[to] == 0){
          p[to] = v;
          dfs (to);
     }
     else if (used[to] == 1&&to!=p[v])
          cycle=true;
}
  used[v] = 2;
}
```

Неориентированный граф G с циклом



Цикл 0→1→2

Алгоритм посетит вершину 0: покрасит ее в серый цвет, запишет предком вершины 1 вершину 0;
посетит вершину 1: покрасит ее в серый цвет, запишет предком вершины 2 вершину 1;
попытается пройти в вершину 0, но она уже серая и при этом не была еще отмечена как предок вершины 2 – значит, алгоритм обнаруживает цикл. Переменная cycle становится равной true.

Для вывода вершин цикла необходимо в момент обнаружения цикла (попытка перехода в вершину, находящуюся в обработке) не только изменить значение переменной cycle=true, но и запомнить начало и конец цикла.

```
cycle=true;
cycle_end = v; //Текущая вершина – конец цикла
cycle_st = to; //Вершина, в которую пытаемся перейти  - начало цикла
```

Вывод вершин цикла осуществляется за один проход по массиву предков, начиная с конца цикла.

```
     while (k!=cycle_st){
          cout<<k<<" ";
          k=p[k];
     }
     cout<<cycle_st;
```

Сложность алгоритма оценивается как O(|E|) – достаточно один раз пройти по каждому ребру графа, чтобы обнаружить цикл и закончить алгоритм.

Поиск циклов в ориентированном графе

Для ориентированных графов алгоритм аналогичен, за исключением того, что как только алгоритм пытается посетить вершину, находящуюся в обработке, сразу можно делать вывод о наличии цикла (дополнительных условий проверять не нужно). В основном тексте программы необходимо осуществить серию запусков DFS от каждой непосещенной еще вершины, с целью поиска циклов.

Ориентированный граф G. Поиск цикла

```
void dfs (int v) {
  used[v] = 1;
  for (size_t i=0; i<g[v].size(); ++i) {
      int to = g[v][i];
      if (used[to] == 0) {
            p[to] = v;
            dfs (to);
      }
      else if (used[to] == 1)
            cycle=true;
}
  used[v] = 2;
}
```

Основной текст программы

```
for (int l=0; l<n;++l){
  if (!used[l]) dfs(l);
  if (cycle) {
      cout<<cycle;
      return 0;
  }
}
cout<<0;
```

Серия запусков DFS в основном тексте программы.

Восстановление вершин, образующих цикл делается, как и в случае неориентированного графа по массиву предков. Сложность алгоритма - O(|E|)

Основные свойства графов. Специальные графы

Для решения олимпиадных задач по теме «Графы» необходимо знать их основные свойства и специальные виды графов.

В неориентированном графе G сумма степеней всех вершин равна удвоенному количеству всех ребер. $\sum_{v_i \in V} d(v_i) = 2|E|$. Доказательство следует из того факта, что каждое ребро учитывается в степенях двух вершин, являющихся его концами.

Количество вершин в графе G, имеющих нечетную степень, четно. Доказательство следует из необходимости четности сумм степеней всех вершин.

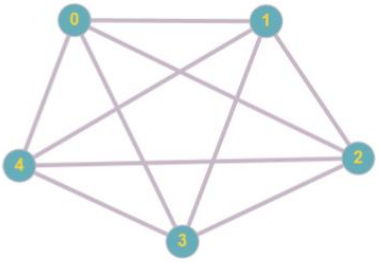
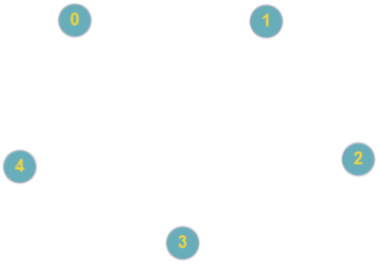
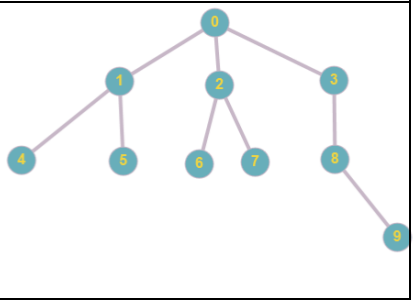
На n вершинах можно построить $2^{C_n^2}$ различных графов. Например, на 3 вершинах можно построить 8 различных графов, включай пустой граф.

Рассмотрим некоторые специальные виды графов.

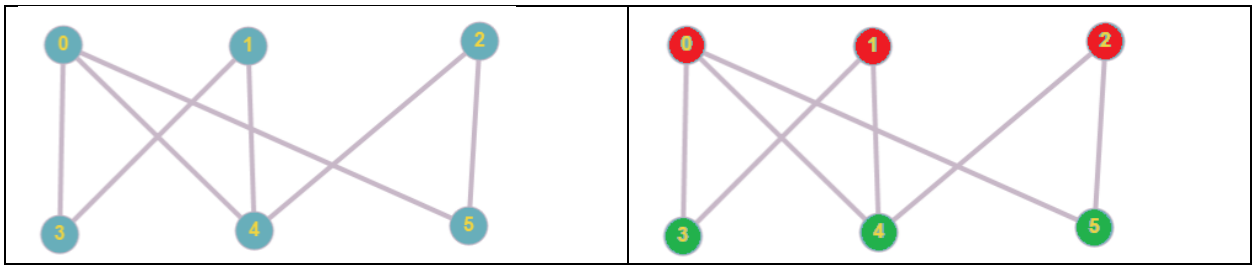
Полный граф K_n – граф, в котором любая его вершина соединена с каждой другой вершиной. У полного графа K_n будет $n(n-1)/2$ ребер.

Пустой граф $\overline{K_n}$ состоит из n изолированных вершин, и является дополнением к полному графу. *Дополнением* $\bar{G}$ к графу G называется граф, у которого множество вершин совпадает с первым графом, а множество ребер графа $\bar{G}$ дополняет множество ребер графа G до полного графа K_n.

Дерево – это простой связный граф без циклов. В дереве, построенном на n вершинах, имеется ровно (n-1) ребро. Можно доказать теорему, полезную для определения того, является ли граф деревом: *Любой простой связный граф, построенный на n вершинах и имеющий (n-1) ребро – дерево*. Примеры графов рассмотренных видов приведены на рисунках.

Полный граф K_n, n=5	Пустой граф $\overline{K_n}$, n=5	Дерево
		

Двудольный граф – граф, вершины которого можно разбить на два множества так, чтобы концы каждого ребра принадлежали различным множествам. На рисунке ниже представлен двудольный граф, состоящий из двух долей: {0, 1, 2} и {3, 4, 5}.



Обратим внимание на тот факт, что каждая доля двудольного графа может быть раскрашена в свой цвет, причем вершины одного цвета не являются смежными.

Полный двудольный граф $K_{m,n}$ – это граф со всеми возможными ребрами между долями.

Проверка графа на двудольность

Алгоритм DFS применяется для проверки графа на двудольность. При обходе графа в глубину делаем раскраску графа в два цвета: {1,2}. Если в процессе работы алгоритма встречаем ребро, концы которого окрашены в один и тот же цвет, то граф является двудольным. Для неориентированного графа требуется проверка, что мы идем не в ту вершину, которая уже помечена как предок рассматриваемой.

Проверка на двудольность неориентированного графа

```
void dfs(int v){
  for(size_t i=0;i<g[v].size();++i){
       int to=g[v][i];
       int try_c=3-used[v];
         if (!used[to]){
              used[to]=try_c;
              p[to]=v;
              dfs(to);
         }
         else if(to!=p[v]&&used[v]==used[to])
                    Two=false;
  }
}
```

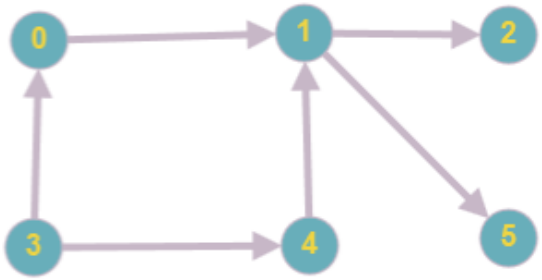
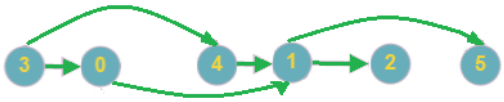
Проверка на двудольность ориентированного графа

```
void dfs(int v){
  for(size_t i=0;i<g[v].size();++i){
       int to=g[v][i];
       int try_c=3-used[v];
         if (!used[to]){
              used[to]=try_c;
              p[to]=v;
              dfs(to);
         }
         else if (used[v]==used[to])
                    Two=false;
  }
}
```

В основном тексте программы цвет стартовой вершины делаем равным единице - used[0]=1 и предполагаем, что граф является двудольным - Two=true. Делаем серию запусков dfs с проверкой на двудольность от каждой непосещенной вершины, начиная с нее как со стартовой. Вывод всех вершин каждой доли можно сделать согласно полученной раскраске вершин в цвета. Сложность алгоритма - O(|E|).

Топологическая сортировка графа

Дан ориентированный граф, не содержащий циклов (предварительно необходимо проверить отсутствие циклов). Топологическая сортировка (topologically sort) упорядочивает вершины графа так, что все ребра идут от вершины с меньшим номером к вершине с большим номером в топологическом порядке.

Ориентированный ациклический граф G	Топологический порядок вершин графа
	3 0 4 1 2 5 – один из возможных топологических порядков. 3 – исток графа, 5 – сток графа. 

На рисунке справа вершины графа G расположены в топологическом порядке. Произведена, так называемая, линеаризация графа. В вершину 3 не входит ни одно ребро – это исток графа. Из вершины 5 не выходит ни одно ребро – это сток графа. У каждого ациклического ориентированного графа есть хотя бы один исток и хотя бы один сток.

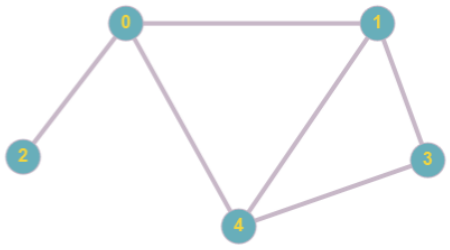
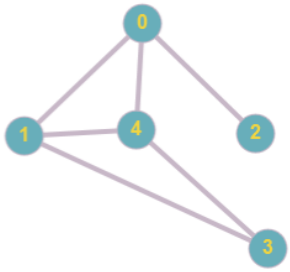
В ациклическом ориентированном графе нет обратных ребер, поэтому для топологической сортировки следует упорядочить вершины в порядке убывания их времени конца обработки вершины time_out[]. То есть, в процедуре DFS при выходе из вершины ее номер добавляется в конец вектора с ответом. Ответ выводится в противоположном порядке. Из основного текста программы выполняется серия запусков DFS от каждой непосещенной вершины.

```
void dfs(int v)//Функция DFS
{
     for (size_t i=0; i< g[v].size(); ++i)
          if (used[g[v][i]]==0)
               dfs(g[v][i]);
     used[v]=2;         //По окончании обработки
     ans.push_back(v); //вершины добавляем ее в вектор ответа
}
int main() //Основной текст программы
{
     /*Считывание данных…*/
     for (int i=0;i<n;++i)
          if (!used[i])
               dfs(i); Запуск от каждой непосещенной вершины
     for (auto i=ans.rbegin();i!=ans.rend();++i)
          cout<<(*i)<<" "; //Вывод вершин ans в обратном порядке
     return 0;
}
```

Одна из распространенных задач на топологическую сортировку заключается в проверке неравенств на непротиворечивость. Есть n переменных, значения которых неизвестны. Известно лишь про некоторые пары переменных, что одна переменная меньше другой. Требуется проверить, не противоречивы ли эти неравенства, и если нет, выдать переменные в порядке их возрастания (если решений несколько — выдать любое). Данная задача решается алгоритмом топологической сортировки графа.

Обход графа в ширину

Обход графа в ширину или поиск в ширину BFS (breadth-first search) является еще одним способом обхода графа. В результате поиска в ширину находится путь кратчайшей длины в невзвешенном графе (ребра которого не имеют весов), то есть путь, содержащий наименьшее число рёбер. Сложность работы алгоритма такая же, как и алгоритма обхода в глубину - $O(|V| + |E|)$.

Неориентированный граф G	Граф G, изображенный в виде слоев по увеличению расстояния от вершины 0
	 Расстояние от вершины 0 до вершин 1, 4, 2 равно 1. Расстояние от вершины 0 до вершины 3 равно 2.

Суть алгоритма заключается в том, чтобы просматривать сначала стартовую вершину 0, затем те вершины, которые удалены от нее на расстояние 1 и так далее слоями: затем просматриваем вершины, которые удалены на расстояние d, далее d+1. Для этого в алгоритме используется очередь Q, в которую сначала заносится стартовая вершина 0. Затем повторяем следующие итерации: пока очередь не пуста, достаем из ее головы очередную вершину и просматриваем всех соседей этой вершины, и если какие-то из них еще не помещены в очередь, то помещаем их в конец очереди. При таком обходе, когда очередь станет пуста, все вершины будут просмотрены, причем в порядке увеличения расстояния от стартовой вершины. Длины кратчайших путей считаются в процессе алгоритма при помощи массива расстояний d[] и массива предков вершин p[].

Приведем алгоритм BFS (источник – сайт http://e-maxx.ru/algo/bfs)

```
queue<int> q; // Очередь вершин
q.push (s); //Добавляем стартовую вершину
vector<bool> used (n); //Вектор признака посещенности вершин
vector<int> d (n), p (n); //Вектор расстояний и предков
used[s] = true; //Стартовую вершину считаем посещенной
p[s] = -1; //У стартовой вершины нет предка
while (!q.empty()) { //Пока очередь не пуста
     int v = q.front(); //Извлекаем из головы очереди вершину
     q.pop(); //удаляем извлеченную вершину
     for (size_t i=0; i<g[v].size(); ++i) { //Просмотр всех
          int to = g[v][i];                 //смежных вершин
          if (!used[to]) {         //Если вершина не посещена,
               used[to] = true; //посещаем ее
               q.push (to); //и добавляем к концу очереди
               d[to] = d[v] + 1;//Считаем расстояние до вершины
               p[to] = v; //Запоминаем предка
          }
     }
}
```

В основном тексте программы для вывода пути до какой-то вершины to, делаем это следующим образом: пути нет, если вершина осталась непосещенной.

```
if (!used[to])
     cout << "No path!";
```

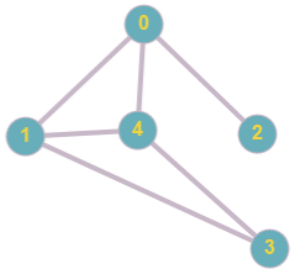
Если путь есть, то его можно восстановить в обратном порядке, используя массив предков и записывая все вершины в массив path[].

```
else {
     vector<int> path;
     for (int v=to; v!=-1; v=p[v])
          path.push_back (v);
```

Разумеется, путь будем выводить, начиная со стартовой вершины. Для этого выводим вектор path[] в обратном порядке.

```
     reverse (path.begin(), path.end());
     cout << "Path: ";
     for (size_t i=0; i<path.size(); ++i)
          cout << path[i] + 1 << " ";
}
```

Для понимания работы алгоритма рассмотрим состояние очереди Q при посещении каждой вершины.

Граф G, изображенный в виде слоев по увеличению расстояния от вершины 0		Порядок посещения вершин	Состояние очереди
	Расстояние от вершины 0 до вершин 1, 4, 2 равно 1. Расстояние от вершины 0 до вершины 3 равно 2.	0 1 4 2 3	0 1 4 2 4 2 3 2 3 3 Очередь пуста

Этот алгоритм можно применить также и к ориентированному графу. Расстоянием между вершинами в этом случае будет также минимальное количество ребер, которое надо пройти от стартовой вершины до данной, в направлении ориентации ребер. При этом расстояние не будет симметрично.

Алгоритм Дейкстры для определения расстояний во взвешенном графе

Применяется для нахождения кратчайших путей от одной вершины (стартовой) до всех остальных вершин в неориентированном (или ориентированном) взвешенном графе, при условии, что все ребра в графе имеют неотрицательные веса. Алгоритм назван в честь голландского ученого Эдсгера Дейкстры и был предложен в 1959 году.

Сложность алгоритма оценивается как $O(|V|^2 + |E|)$

Обход в ширину применяется для определения расстояний между вершинами в том случае, когда длина каждого ребра одинакова (невзвешенные графы). На практике более распространена ситуация, когда каждое ребро имеет определенную длину, то есть ребра взвешены. Примером может служить карта местности с расстояниями между городами, такая, как например, на рисунке ниже. Требуется уметь определять наикратчайшее расстояние между любой парой вершин. Рассмотрим подробнее алгоритм Дейкстры, который применяется именно для решения таких задач.

Взвешенный граф удобно хранить в виде списка смежности, где для каждой вершины определен вектор пар – (смежная вершина, расстояние до этой вершины). vector<vector<pair<int,int>>> g – список смежности графа.

Карта Липецкой области

Таблица расстояний между некоторыми городами Липецкой области

Липецк – Елец	85
Липецк – Грязи	31
Липецк – Лебедянь	62
Липецк – Задонск	89
Липецк – Данков	85
Липецк – Усмань	75
Липецк – Хлевное	64

Описание алгоритма

Граф G.

Количество вершин – n.

Количество ребер – m.

s – стартовая вершина

d[] – массив текущих кратчайших расстояний из вершины s в вершины v_i.

d[s]=0, d[v]=∞ - начальные присваивания.

used[] – массив посещения вершин. Изначально все вершины не посещены.

В алгоритме n итераций

1. На очередной итерации выбирается вершина v_i с наименьшей величиной d[v_i] и еще не пройденная ранее. Выбранная вершина v_i отмечается посещенной.
2. Просматриваются все вершины, исходящие из вершины v_i, и для каждой такой вершины to алгоритм пытается улучшить значение d[to] по формуле d[to]=min(d[to],d[v]+len), где len – расстояние от вершины v до to
3. Пункты 1-3 алгоритма повторяются

Реализация

```
const int inf=1e9;
int  main(){
  int n;
  cin>>n;
  vector<vector<pair<int,int>>> g(n);
  // чтение графа…
  vector <int> d (n, inf), p(n);
  vector<int> u(n);
  int s; - стартовая вершина
  d[s]=0;
  for (int i=0; i<n;++i) {
     int v=-1;
     for (int j=0; j<n;++j)
     if (!u[j]&&(v==-1||d[j]<d[v]))
          v=j;
          if (d[v]==inf) break;
          u[v]=true;
  for (size_t  j=0; j<g[v].size();++j){
     int to=g[v][j].first;
     int len =g[v][j].second;
     if (d[v]+len<d[to]){
          d[to]=d[v]+len;
          p[to]=v; }
     }
  }
}
```

Вам предоставляется возможность проделать практическую работу по приведенному примеру и определить какое же кратчайшее расстояние между Липецком и всеми другими городами. Разумеется, нужно узнать не только кратчайшее расстояние, но и путь, по которому от Липецка до каждого города оно достигается.

Вопросы

1. В графе G n вершин и $\Delta(G)=k$. Какое наибольшее количество компонент связности может быть в G?
2. Сколько ребер в графе $\overline{K_{m,n}}$? Запишите формулу для любых n и m.
3. Эйлеровым циклом в неориентированном графе называется замкнутый путь, проходящий по всем ребрам графа ровно по одному разу. Докажите, что в неориентированном графе эйлеров цикл есть тогда и только тогда, когда он связен и степени всех вершин четны.
4. Доказать, что неориентированный граф является двудольным тогда и только тогда, когда в нем нет циклов нечетной длины.
5. Рассмотрим граф G (V,E), имеющий V вершин и E ребер. **Раскраской графа G** называется окрашивание вершин графа G такое, что никакие две смежные вершины не имеют одинаковый цвет. **Хроматическое число графа** $\chi(G)$ - это наименьшее число цветов, которое используется для раскраски графа. Известен жадный алгоритм раскраски графа.

   Жадный алгоритм последовательного раскрашивания

   Входные данные: граф G (V,E)

   Выходные данные: массив c[v] раскрашенных вершин

   Шаг 1. Для всех вершин определить множество A ={1,2,3..,n} всех цветов.

   Шаг 2. Выбрать стартовую вершину (с которой начинаем алгоритм). Раскрасить вершину в цвет color. Вычеркнуть этот цвет из множества цветов всех вершин, смежных со стартовой.

   Шаг 3. Выбрать не раскрашенную вершину v

   Шаг 4. Раскрасить выбранную вершину в минимально возможный цвет из множества A. Вычеркнуть этот цвет из множества цветов всех вершин, смежных с вершиной v.

   Шаг 5. Проделать шаг 3, шаг 4 для всех нераскрашенных вершин графа.

   На основе этого алгоритма раскрасьте выбранный вами граф из данной лекции, так, чтобы получилась правильная раскраска.
6. Составьте трассировочную таблицу по алгоритму Дейкстры к графу, приведенному в лекции.
7. В лекции не был рассмотрен алгоритм поиска компонент сильной связности ориентированного графа. **Компонентой сильной связности** (strongly connected component) называется такое (максимальное по включению) подмножество вершин графа, что любые две вершины этого подмножества достижимы друг из друга. Проиллюстрируйте на рисунке ориентированного графа наличие у него компонент сильной связности. Подумайте над реализацией данного алгоритма.

Литература

Сайт MAXimal. Проверка графа на ацикличность и нахождение цикла. Электронный источник. http://e-maxx.ru/algo/finding_cycle

Дасгупта С. Алгоритмы / С. Дасгупта, Х. Пападимитриу, У. Вазирани; Пер. с англ.под ред.А.Шеня. – М.МЦНМО, 2014

Основы теории графов. Академический Университет (СПбАУ), Computer Science Center (CS центр). Курс Александра Омельченко. Сайт stepic.org

Окулов С.М. Программирование в алгоритмах /С.М.Окулов. – М.: БИНОМ. Лаборатория знаний, 2002.

Дискретная математика. Курс А.Дайняк. Сайт stepic.org