Министерство образования Республики Беларусь
Учреждение образования
«Белорусский государственный университет
информатики и радиоэлектроники»

Кафедра информатики

С. В. Актанорович, **А. А. Волосевич**, **С. И. Сиротко**

АЛГОРИТМЫ И СТРУКТУРЫ ДАННЫХ. ПОТОКОВЫЕ АЛГОРИТМЫ

Методическое пособие
по курсу «Теория графов. Потоковые алгоритмы»
для студентов специальности I-31 03 04 «Информатика»
всех форм обучения

Минск 2011

УДК [519.178+004.043](076)
ББК 22.176я73
А 43

Р е ц е н з е н т :
заведующий кафедрой ПОИТ БГУИР,
кан. тех. наук В.В. Бахтизин

Актанорович, С. В.

А 43 Алгоритмы и структуры данных. Потоковые алгоритмы : метод. пособие по курсу «Теория графов. Потоковые алгоритмы» для студ. спец. I-31 03 04 «Информатика» всех форм обуч. / С. В. Актанорович, А. А. Волосевич, С. И. Сиротко. – Минск: БГУИР, 2011. – 47 с.: ил.
ISBN 000-000-000-000-0

Методическое пособие составлено в соответствии с рабочей программой курса «Теория графов. Потоковые алгоритмы». В него включены базовые определения теории графов и основные результаты теории потоков на взвешенных однородных сетях. Описаны алгоритмы нахождения максимального потока в сети и максимального паросочетания в двудольном графе. Приводится реализация полученных алгоритмов на языке программирования C++.

Пособие может быть рекомендовано студентам и магистрантам технических специальностей для изучения основ теории потоков.

УДК [519.178+004.043](076)
ББК 22.176я73

ISBN 000-000-000-000-0

Содержание

Введение

Теория графов находит широкое применение в большом количестве прикладных задач и алгоритмы обработки графов очень важны. В данном пособии рассматривается задача о максимальном потоке на взвешенном графе, а также задача поиска наибольшего паросочетания в двудольном графе. Задача нахождения максимального потока в сети довольно часто возникает на практике, а также является подзадачей при решении более крупного класса задач [1]. К тому же, многие задачи комбинаторной оптимизации могут быть сформулированы в терминах задачи о максимальном потоке. Примером может быть упоминавшаяся выше задача поиска наибольшего паросочетания[1, 2].

Данное пособие является кратким введением в теорию решения задач, связанных с нахождением максимального потока в сети. В первую очередь оно предназначено для тех, кто занимается алгоритмическим программированием. Авторами сознательно опущены многие доказательства теорем, на которые опираются алгоритмы решения тех или иных задач. В то же самое время делается акцент на практической реализации полученных теоретических результатов современной дискретной математики (в частности, исследования операций и комбинаторной оптимизации). Для каждого из рассмотренных алгоритмов приведены примеры, предлагавшиеся на различных соревнованиях по программированию. Таким образом, целью данного пособия является приведение алгоритмов, решающих классические задачи теории графов, с их корректной реализацией на одном из высокоуровневых языков программирования. Такими языками могут выступать C++, Java, Delphi, C# и др. [3]

Пособие построено следующим образом. Вначале дается неформальная постановка рассматриваемых задач, после чего вводится необходимый ряд определений. Далее излагаются математические факты, на базе которых строятся алгоритмы решения сформулированных проблем. В конце каждого из разделов приводятся задачи, решения которых опираются на теорию решения задач, рассматриваемого раздела.

Предполагается, что читатель знаком с основами теории графов, а также владеет базовыми навыками программирования на языке C++. Язык C++ был выбран по причине его высокой популярности в среде «олимпиадников»[3]. Перевод же реализаций алгоритмов на другие языки программирования не составляет особого труда.

1. Основные понятия теории графов

В данном разделе формально изложены основные определения, необходимые для работы с графами [1,4].

Определение 1. *Ориентированным графом* называется пара (V,E), где V – непустое конечное множество, а E – бинарное отношение на V, т.е. подмножество множества $V \times V$. Множество V называют *множеством вершин* графа. Множество E называют *множеством ребер* графа.[1]

Определение 2. В *неориентированном* графе $G=(V,E)$ множество ребер E состоит из *неупорядоченных* пар вершин (u,v), где $u,v \in V$. Для неориентированного графа (u,v) и (v,u) обозначают одно и то же ребро.

При изображении графов будем обозначать направленное ребро стрелкой, а ненаправленное – отрезком.

Многие понятия параллельно определяются для ориентированных и неориентированных графов. Про ребро (u,v) ориентированного графа говорят, что оно выходит из вершины u и входит в вершину v. Про ребро (u,v) неориентированного графа говорят, что оно *инцидентно* вершинам u и v.

Определение 3. *Степенью вершины* в неориентированном графе называется число инцидентных ей ребер. Для ориентированного графа различают *исходящую степень*, определяемую как число выходящих из вершины ребер, и *входящую степень*, определяемую как число входящих в вершину ребер. Сумма исходящей и входящей степеней называется *степенью* вершины.

Степень вершины v будем обозначать через $\deg(v)$. Исходящую степень вершины обозначим через $\deg^-(v)$, а входящую – через $\deg^+(v)$.

Определение 4. *Путь длины* k из вершины u в вершину v определяется как последовательно вершин $(v_o, v_1, \ldots, v_k)$, в которой $v_o = u$, $v_k = v$ и $(v_{i-1}, v_i) \in E$ для всех $i = 1,2, \ldots, k$.

Определение 5. Путь называется *простым*, если все вершины этого пути различны. Простой путь содержит попарно различные ребра.

Определение 6. *Цепью длины* k назовём некоторый простой путь, содержащий ровно k рёбер.

Определение 7. *Подграфом* графа $G=(V,E)$ называется граф $G'=(V',E')$ такой, что $E' \subseteq E$, $V' \subseteq V$.

Определение 8. Неориентированный граф называется *связным*, если для любых двух вершин существует путь между ними.

Определение 9. Связный граф $G=(V,E)$ называется *двудольным*, если множество вершин V можно разбить на две части V_1 и V_2 такие, что концы любого ребра $(u,v) \in E$ находятся в разных частях.

Каждой вершине и каждому ребру графа может быть приписан некоторый вес. Это значит, что существует весовая функция, отображающая множест-

[1] Иногда ребра ориентированного графа называют *дугами.*

во ребер и/или вершин на некоторое множество весов. Говоря о весе ребра или вершины, подразумевается значение весовой функции на соответствующем ребре или вершине.

Определение 10. *Сетью* называется ориентированный граф $G=(V,E)$, каждому ребру $(u,v)\in E$ которого поставлено в соответствие число $c(u,v)\geq 0$, называемое пропускной способностью ребра. В случае, если ребро $(u,v)\notin E$ пропускная способность ребра (u,v) полагается равной 0.

Следует сказать, что пропускная способность ребер сети может быть установлена в бесконечность. С точки зрения ЭВМ это означает, что пропускная способность таких ребер будет равна заведомо большому числу, которое превышает пропускные способности оставшихся ребер сети

Если в сети заданы пропускные способности на вершины, то такие вершины разделяются на две, с целью преобразования исходной сети к сети, в которой функция пропускной способности c задается только на множестве ребер. Здесь следует отметить, что большинство алгоритмов, разработанных в рамках теории потоков, оперируют именно такими функциями[1,5].

Пример разделения вершины приведен на рис. 1.

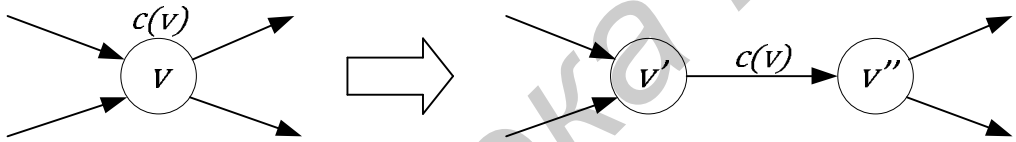


Рис. 1. Разделение вершин.

Определение 11. *Истоком* называется любая вершина $v\in V$ заданной сети $G=(V,E)$ такая, что $\deg^-(v)>0$, а $\deg^+(v)=0$.

Определение 12. *Стоком* называется любая вершина $v\in V$ заданной сети $G=(V,E)$ такая, что $\deg^-(v)=0$, а $\deg^+(v)>0$.

Пример сети приведен на рис. 2.

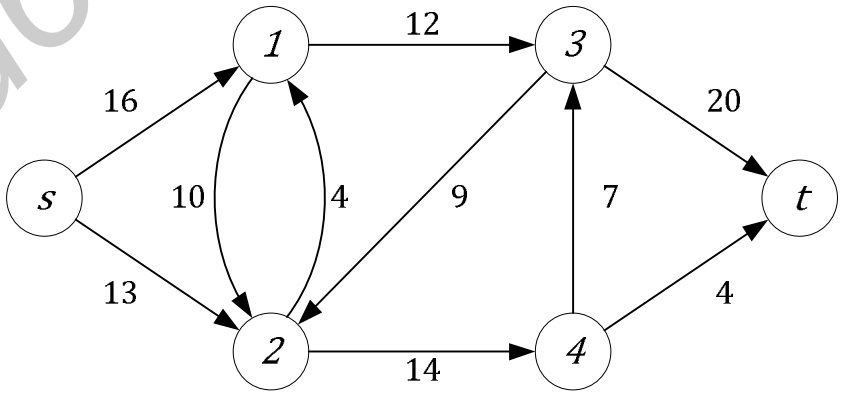


Рис. 2. Пример сети с истоком s и стоком t.

Если в сети можно выделить более одного истока и стока, то такая сеть легко преобразуется к сети с одним истоком и одним стоком путем добавления дополнительных вершин. Очень часто такие исток и сток называют *супер-истоком* и *супер-стоком* соответственно. Пример такого преобразования приведен на рис. 3.

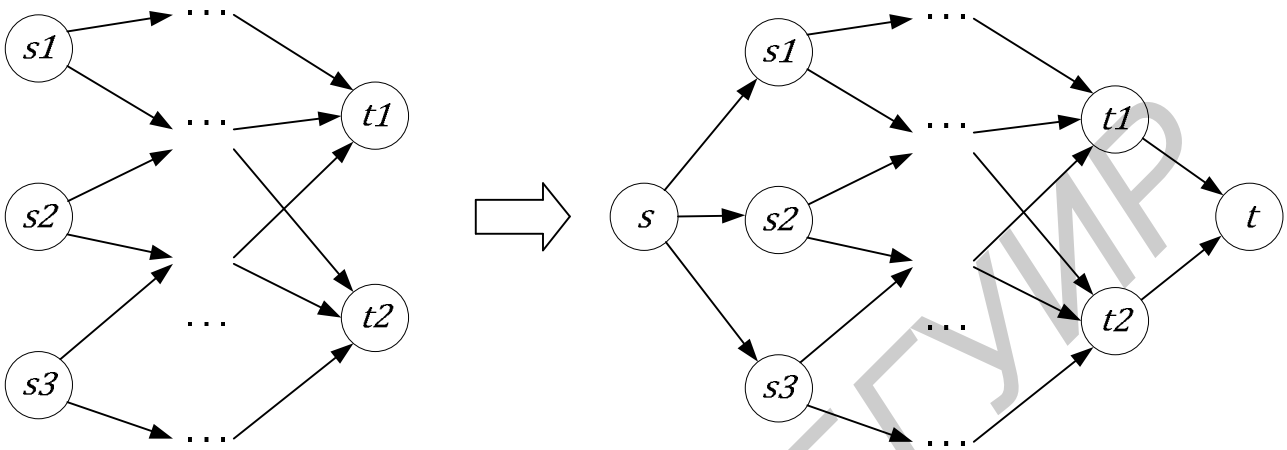


Рис. 3. Сеть со многими истоками и стоками.
Здесь s – супер-сток, t – супер-исток.

Ключевым понятием теории потоковых алгоритмов и, в целом, теории сетевых моделей, является понятие разреза[1,5].

Определение 13. *Разрезом* сети $G=(V,E)$ с истоком s и стоком t называется пара множеств V_s и V_t, удовлетворяющая следующим условиям

- $s \in V_s,\ t \in V_t$;
- $V_s \bigcup V_t = V$;
- $V_s \bigcap V_t = \varnothing$.

Определение 14. *Пропускной способностью* разреза (V_s, V_t) называется сумма $c(V_s, V_t)$ пропускных способностей дуг (u,v), для которых выполнено условие $u \in V_s,\ v \in V_t$.

Пример разреза для сети рис. 2 приведен на рис. 4.

Формально пропускную способность разреза $c(V_s, V_t)$ можно описать следующим выражением

$$c(V_s, V_t) = \sum_{u \in V_s,\ v \in V_t} c(u,v).$$

Таким образом, для примера рис. 4

$$c(V_s, V_t) = c(1,3) + c(2,4) = 12 + 14 = 26.$$

Следует отметить, что пропускная способность разреза не учитывает пропускных способностей дуг, ведущих из множества V_t во множество V_s. Такой дугой на рис. 5 является дуга $(3,2)$.

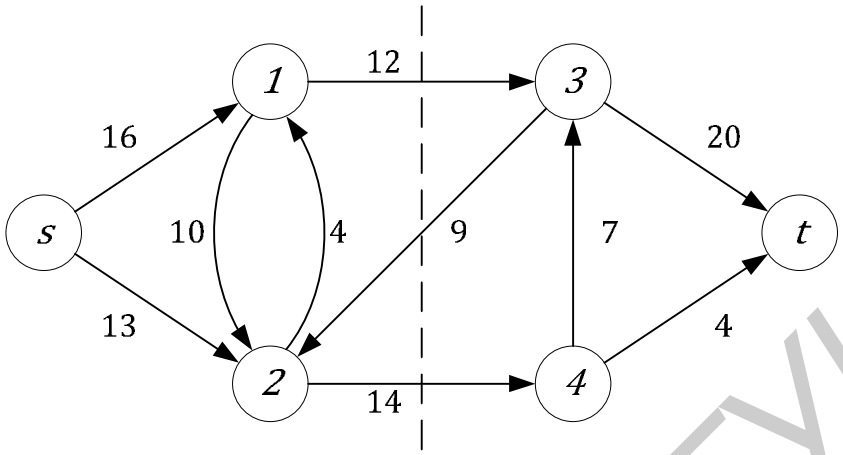


Рис. 4. Разрез (V_s, V_t) в сети рис. 3.
Здесь $V_s = \{s,1,2\}$ и $V_t = \{3,4,t\}$.

Определение 15. *Минимальным разрезом* сети $G = (V, E)$ называется такой разрез, чья пропускная способность минимальна из всех разрезов (V_s, V_t) данной сети.

Для примера рис. 4 минимальным разрезом будет разрез (V_s, V_t), где $V_s = \{s,1,2,4\}$ и $V_t = \{3,t\}$. В этом случае

$$c(V_s, V_t) = c(1,3) + c(4,3) + c(4,t) = 12 + 7 + 4 = 23.$$

В следующем разделе дается краткое описание методов хранения графов, позволяющих эффективно оперировать введенными выше понятиями.

2. Структуры данных и базовые алгоритмы

Графы можно представить в памяти компьютера несколькими способами. Пусть задан граф $G = (V, E)$, в котором n вершин и m ребер.

1. *Матрица смежности.* Граф G можно представить, используя матрицу M размерности $n \times n$, в которой элемент $M[i, j]$ равен 1, если $(i, j) \in E$ и 0 в противном случае. Если на ребрах задана весовая функция c, то в ячейках матрицы можно хранить значения $c(i, j)$. Если граф хранится в виде матрицы смежности, то становится возможным быстро отвечать на вопрос, являются ли вершины смежными. В то же самое время, матрицы смежности не лучшим образом подходят для хранения разреженных графов и мультиграфов[2]. Следует отметить, что матрицы смежности достаточно ресурсоемки по памяти: хранение графа с количеством вершин n требует $O(n^2)$ ячеек памяти.

2. *Списки смежных ребер.* Суть данного метода заключается в хранении для каждой вершины графа списка инцидентных с ней ребер. Если ребро является неориентированным, то оно будет учтено в списках смежности обеих вершин. Следует отметить, что данный способ хранения графов может быть реализован как посредством массива, так и посредством указателей. Оба подхода требуют $O(n + m)$ ячеек памяти.

Для хранения графа в виде списков смежных ребер будем использовать структуру, представленную в листинге 1.

Листинг 1. Структура данных для представления ребра сети.

```
1.  struct tedge
2.  {
3.      int source, destination;
4.      int capacity, flow;
5.      tedge *next, *reverse;
6.      tedge(int source, int destination, int capacity)
7.      {
8.          this->source = source;
9.          this->destination = destination;
10.         this->capacity = capacity;
11.         this->flow = 0;
12.         this->next = this->reverse = 0;
13.     }
14. }
```

Сам же граф можно описать массивом, где *max_nvertices* – максимальное число вершин в графе. Операции добавления направленного ребра и ненаправленного ребра в сеть представлены в листинге 2.

[2] *Мультиграфом* называется граф, в котором разрешено наличие нескольких ребер между фиксированными парами вершин.

Листинг 2. Структура данных для представления графа. Операции добавления ориентированных и неориентированных ребер в граф.

```
1.  typedef tedge *tgraph[max_nvertices + 1];
2.  typedef vector<tedge*> tpath;
3.
4.  /* добавление дуги (source, destination) */
5.  void add_arc(tgraph &g, int source, int destination, int capacity)
6.  {
7.      tedge *x = new tedge(source, destination, capacity);
8.      x->next = g[source];
9.      g[source] = x;
10. }
11.
12. /* добавление ребра (source, destination) */
13. void add_edge(tgraph &g, int source, int destination, int capacity)
14. {
15.     add_arc(g, source, destination, capacity);
16.     add_arc(g, destination, source, 0);
17.     g[source]->reverse = g[destination];
18.     g[destination]->reverse = g[source];
19. }
```

Стоит обратить внимание на строку 16, в которой добавляется дуга с пропускной способностью равной 0. Почему так происходит будет пояснено в следующем разделе.

Если необходимость в использовании динамических структур данных отсутствует, то можно обойтись обычными массивами, на базе которых и будут строиться списки смежности. Детали реализации приведены в листинге 3.

Листинг 3. Реализация списков смежности через массивы.

```
1.  int head[max_nvertices], next[max_nedges];
2.  int source[max_nedges], destination[max_nedges];
3.  int capacity[max_nedges], flow[max_nedges], reverse[max_nedges];
4.  int nedges = 0;
5.
6.  void add_arc(int source, int destination, int capacity)
7.  {
8.      ::source[nedges] = source;
9.      ::destination[nedges] = destination;
10.     ::capacity[nedges] = capacity, flow[nedges] = 0;
11.     next[nedges] = head[source];
12.     head[source] = nedges;
13.     nedges = nedges + 1;
14. }
```

Перед непосредственным использованием данной реализации списков массив *head* стоит инициализировать значением *undefined* (например, −1).

Основными алгоритмами обработки графов служат алгоритмы поиска в ширину и в глубину. Суть каждого из этих методов состоит в просмотре вершин заданного графа со сбором необходимой статистической информации[3]. Базовые реализации этих алгоритмов приведены в листингах 4, 5 [4,6].

Листинг 4. Процедура поиска в ширину.

```
1.  void bfs(tgraph &g, int nvertices, int source, int sink)
2.  {
3.      for (int i = 1; i <= nvertices; ++i)
4.      {
5.          visited[i] = false;
6.      }
7.      queue<int> q;
8.      for (q.push(source); !q.empty(); q.pop())
9 .     {
10.         int v = q.front();
11.         visited[v] = true;
12.         for (tedge* x = g[v]; x != 0; x = x->next)
13.         {
14.             if (!visited[x->destination])
15.             {
16.                 q.push(x->destination);
17.             }
18.         }
19.     }
20. }
```

Листинг 5. Рекурсивная процедура поиска в глубину.

```
1.  void dfs(tgraph &g, int nvertices, int source, int sink)
2.  {
3.      for (int i = 1; i <= nvertices; ++i)
4.      {
5.          visited[i] = false;
6.      }
7.      dfs(g, source, sink);
8.  }
9.
10. void dfs(tgraph &g, int source, int sink)
11. {
12.     visited [source] = true;
13.     for (tedge *x = g[source]; x != 0; x = x->next)
14.         if (!visited[x->destination])
15.             dfs(g, x->destination, sink);
16. }
```

[3] Такой информацией может быть время посещения вершины, количество входящих и исходящих ребер, глубина вершины в остовном дереве и т.д.[6].

Следует отметить тесную связь этих алгоритмов, так как каждый из них можно получить путем спецификации следующей подпрограммы, которая в качестве параметра принимает контейнер для хранения вершин, листинг 6.

Листинг 6. Обобщенная процедура обхода графа.

```
1.  void traversal(tgraph &g, int nvertices, int source, int sink,
2.                 tcontainer<int> &container)
3.  {
4.      for (int i = 1; i <= nvertices; ++i)
5.      {
6.          visited[i] = false;
7.      }
8.      visited[source] = true;
9.      for (container.push(source); !container.empty(); container.pop())
10.     {
11.         int v = container.front();
12.         for (tedge* x = g[v]; x != 0; x = x->next)
13.         {
14.             if (!visited[x->destination])
15.             {
16.                 container.push(x->destination);
17.             }
18.         }
19.     }
20. }
```

Тогда реализация поиска в ширину, т.е. процедура *bfs()*, получается из процедуры *traversal()* путем передачи в нее контейнера *queue* (либо структуры данных, работающей по принципу FIFO)[6]. Обход графа в глубину, аналогичный тому, который совершает процедура *dfs()*, моделируется, если в качестве контейнера для хранения вершин используется контейнер *stack()* (либо структура данных, работающая по принципу LIFO)[6]. Стоит обратить внимание на такую реализацию поиска в глубину, т.к. при довольно большом числе вершин (до 100,000 включительно) могут возникать ошибки переполнения, связанные с рекурсивными вызовами.

3. Максимальный поток в сети

Задача о максимальном потоке для данной сети неформально может быть сформулирована следующим образом: «Задана сеть труб и для каждой трубы известна ее пропускная способность. Трубы соединены в своих концевых точках. Какое максимальное количество воды можно пропустить из заданной начальной точки в заданную конечную точку?».

Будем считать, что в вершинах вещество не накапливается – сколько приходит, столько и уходит (если только вершина не является стоком или истоком). Это свойство называется *законом сохранения потока* [4].

Перед тем как перейти к непосредственному рассмотрению методов решения таких типов задач, введем ряд важных определений.

3.1. Основные понятия

Центральным понятием данного раздела является понятие потока.

Определение 16. Пусть задана сеть $G=(V,E)$ с истоком s и стоком t, пропускная способность которой задается функцией c. *Потоком* в сети G назовем функцию $f: V \times V \to \boldsymbol{R}$, обладающую следующими свойствами

- $f(u,v) \le c(u,v), \forall u,v \in V$;
- $f(u,v) = -f(v,u), \forall u,v \in V$;
- $\sum_{v \in V} f(u,v) = 0$, $\forall u \in V / \{s,t\}$.

Величина $f(u,v)$ может быть как положительной, отрицательной, так и равной 0. Она определяет, сколько единиц вещества движется из вершины u в вершину v (отрицательные значения соответствуют движению в обратную сторону). Пример потока в сети рис. 2. приведен на рис. 5.

Определение 17. *Величина* потока f в сети $G=(V,E)$ с истоком s и стоком t определяется как сумма

$$\sum_{v \in V} f(s,v)$$

и обозначается $|f|$.

Задача о максимальном потоке для заданной сети $G=(V,E)$ с истоком s и стоком t состоит в нахождении потока максимальной величины.

Рассмотрим ряд определений, связанных с понятием разреза.

Определение 18. Величина *потока через разрез* (V_s, V_t) в сети $G=(V,E)$ для заданного потока f определяется как сумма $f(V_s, V_t)$ по пересекающим разрез ребрам.

Формально величину потока через разрез можно записать в виде следующего выражения

$$f(V_s, V_t) = \sum_{(u,v) \in (V_s, V_t)} f(u,v) - \sum_{(u,v) \in (V_t, V_s)} f(u,v).$$

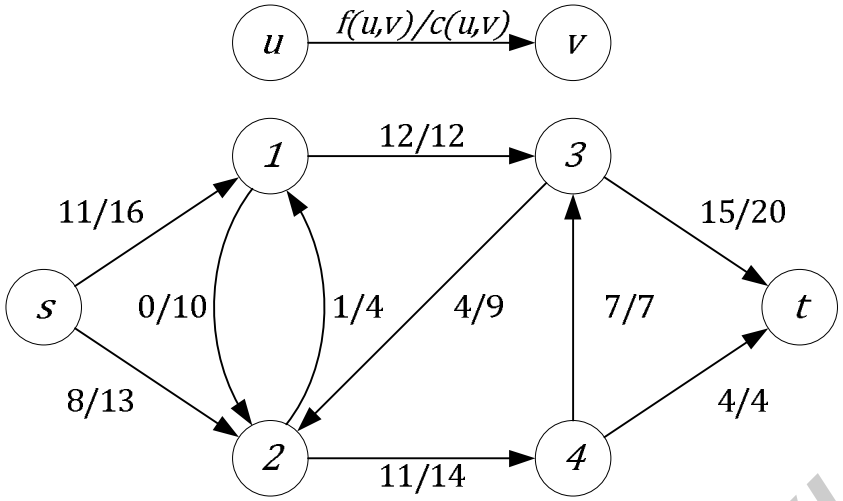


Рис. 5. Пример потока f в сети $G=(V,E)$ величины 19.

Для примера рис. 5 получим

$$f(V_s, V_t) = f(1,3) + f(2,4) - f(3,2) = 12 + 11 - 4 = 19.$$

Определение 19. Пусть f – поток в сети $G=(V,E)$. *Остаточной пропускной способностью* ребра $(u,v) \in E$ называется величина $c_f(u,v)$, определяемая как

$$c_f(u,v) = c(u,v) - f(u,v).$$

Определение 20. Пусть f – поток в сети $G=(V,E)$. Назовем сеть $G_f=(V,E_f)$, где

$$E_f = \{(u,v) \in V \times V : c_f(u,v) > 0\},$$

остаточной сетью сети G, порожденной потоком f.

Пример остаточной сети для сети рис. 5 приведен на рис. 6.

Определение 21. Пусть f – поток в сети $G=(V,E)$. *Дополняющим путем* в остаточной сети G_f назовем простой путь из истока s в сток t.

Перед тем как перейти к непосредственному рассмотрению алгоритмов нахождения максимального потока сделаем ряд предположений, в контексте которых рассматриваемые ниже алгоритмы и будут разрабатываться[1].

Предположение 1. При рассмотрении сетей будем предполагать, что каждая вершина $v \in V$ лежит на каком-то пути $s \rightsquigarrow v \rightsquigarrow t$ из истока в сток. Если это так, то рассматриваемый граф является связным.

Предположение 2. Пропускные способности ребер являются целыми положительными числами. Это ограничение не является строгим, так как современные ЭВМ хранят числа в вещественном формате. Преобразование же вещественных чисел в целые осуществляется путем операции масштабирования (т.е., умножения на достаточно большое число).

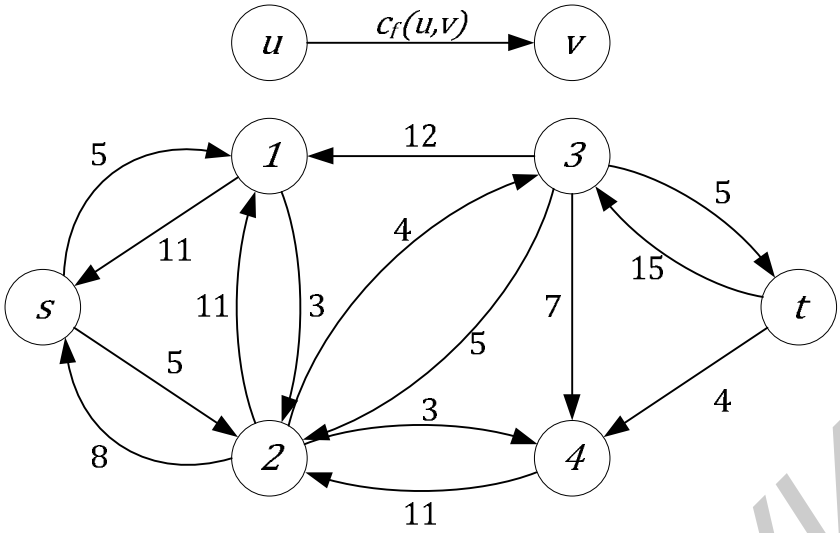


Рис. 6. Остаточная сеть G_f для сети G рис. 5.

Предположение 3. Заданная сеть содержит только дуги. Неориентированное же ребро можно преобразовать в два ориентированных следующим образом: вместо ребра (u,v) с пропускной способность $c(u,v)$ в сети появятся дуга (u,v) с пропускной способность $c(u,v)$ и дуга (v,u) с пропускной способностью равной 0 (рис. 7).

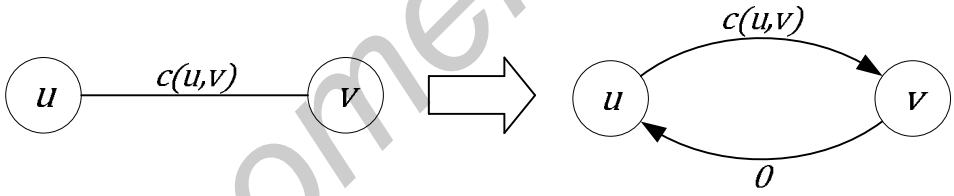


Рис. 7. Преобразование неориентированного ребра
к двум ориентированным.

Предположение 4. Заданная сеть не содержит ориентированного пути из истока s в сток t, состоящего из дуг с бесконечными пропускными способностями. Если же такой путь найдется, то величина максимального потока будет неограниченной сверху.

Предположение 5. Если в сети присутствует дуга (u,v), то в ней же присутствует и дуга (v,u). Это ограничение не является строгим в силу того, что в сети разрешено наличие ребер с нулевой пропускной способностью.

Перед непосредственным рассмотрением алгоритмов нахождения максимального потока в заданной сети G, приведем код функции *get_flow()*, которая возвращает текущий поток в G, листинг 7. Реализация функции *get_flow()* базируется на определении 17.

Листинг 7. Функция get_flow().

```
1. int get_flow(tgraph &g, int source, int sink)
2. {
3.     int result = 0;
4.     for (tedge *x = g[source]; x != 0; x = x->next)
5.     {
6.         result = result + x->flow;
7.     }
8.     return result;
9. }
```

В следующем разделе рассматривается общий алгоритм дополняющего пути, который опирается на идеи, изложенные выше.

3.2. Алгоритм дополняющего пути

Базовым алгоритмом нахождения максимального потока в заданной сети $G = (V, E)$, является общий алгоритм дополняющего пути, реализация которого приведена в листинге 8.

Листинг 8. Алгоритм дополняющего пути.

```
1. int generic_augmenting_path_algorithm(tgraph &g, int nvertices,
2.                                       int source, int sink)
3. {
4.     while (augment(g, nvertices, source, sink))
5.     {
6.         tpath path = get_augmenting_path(source, sink);
7.         int delta = +oo;
8.         for (size_t i = 0; i < path.size(); ++i)
9.         {
10.            tedge *x = path[i];
11.            delta = min(delta, x->capacity – x->flow);
12.        }
13.        for (size_t i = 0; i < path.size(); ++i)
14.        {
15.            tedge *x = path[i];
16.            x->flow += delta;
17.            if (x->reverse != 0)
18.            {
19.                x->reverse->flow -= delta;
20.
21.        }
22.    }
23.    return get_flow(g, source, sink);
24. }
```

Идея алгоритма дополняющего пути основана на следующей теореме, сформулированной и доказанной Фордом и Фалкерсоном в 1956 году[7].

Теорема 1. Пусть f – поток в сети $G = (V, E)$. Тогда следующие утверждения равносильны

- Поток f максимален в сети G.
- Остаточная сеть G_f не содержит дополняющих путей.
- Для некоторого разреза (V_s, V_t) сети G выполнено равенство

$$| f | = c(V_s, V_t).$$

Первой фазой алгоритма является проверка существования дополняющего пути в заданном графе (строка 4). Проверить наличие такого пути можно при помощи процедуры поиска в глубину. Пример реализации функции *augment()* приведен в листинге 9.

Листинг 9. Функции augment().

```
1. bool augment(tgraph &g, int nvertices, int source, int sink)
2. {
3.     for (int i = 1; i <= nvertices; ++i)
4.         parent[i] = 0;
5.     return dfs(g, source, 0, sink);
6. }
```

В силу того, что функция *augment()* возвращает булевское значение, то процедура *dfs* (строка 5) требует незначительных изменений (листинг 10).

Листинг 10. Функция dfs().

```
1.  bool dfs(tgraph &g, int source, tedge *by_edge, int sink)
2.  {
3.      parent[source] = by_edge;
4.      if (source == sink)
5.          return true;
6.      for (tedge *x = g[source]; x != 0; x = x->next)
7.          if (parent[x->destination] == 0 &&
8.              x->capacity - x->flow > 0)
9.          {
10.             bool result = dfs(g, x->destination, x, sink);
11.             if (result)
12.                 return true;
13.         }
14.     return false;
15. }
```

Обратим внимание на строки 7 и 8, в которых анализируются ребра графа, выходящие из текущей вершины *source*. Как и в базовой реализации поиска в глубину, в первую очередь проверяется, посещалась ли вершина *destination* ранее. Если нет (*parent[destination] = 0)*, то проводится тест на принадлежность данного ребра остаточной сети (напомним, что ребро принадлежит остаточной сети, если его остаточная пропускная способность строго положительна). Если

тест положителен, то процедура рекурсивно вызывается для вершины *destination*. Если же вершина *destination* посещалась ранее, либо остаточная пропускная способность ребра равна 0, то данное ребро игнорируется и осуществляется анализ следующих ребер из списка смежности ребер вершины *source.* Результат работы процедуры *dfs()* будет положителен только в том случае, если достигнут сток *sink* (строки 4-5). Последнее же выполняется только при условии существования дополняющего пути.

Функция построения дополняющего пути *get_augmenting_path()* вызывается после того, как была осуществлена проверка существования такого пути (листинг 8, строка 6). Логика функции заключается в помещении ребер, просмотренных функцией *dfs()* (листинг 10), в массив *tpath*. Детали реализации функции приведены в листинге 11.

Листинг 11. Функция get_augmenting_path().

```
1.  tpath get_augmenting_path(int source, int sink)
2.  {
3.      tpath result;
4.      for (int i = sink; i != source;)
5.      {
6.          tedge *x = parent[i];
7.          result.push_back(x);
8.          i = x->source;
9.      }
10.     return result;
11. }
```

В заключение раздела 3.2., приведем теорему, устанавливающую некоторые из свойств, которыми обладает приведенный выше алгоритм [1,4].

Теорема 2. Если все пропускные способности дуг заданной сети $G=(V,E)$ являются целыми положительными числами, то существует целочисленный максимальный поток, протекающий в сети G.

Данная теорема не утверждает, что любое оптимальное решение является целочисленным. Задача о максимальном потоке может содержать и нецелочисленные решения и, более того, такие решения всегда существуют. Основным результатом, который устанавливает теорема 2, является наличие хотя бы одного целочисленного решения.

В [7] показано, что алгоритмическая сложность алгоритма Форда-Фалкерсона равняется $O(|E|*|f|)$, где $|E|$ – количество ребер в сети, $|f|$ – величина максимального потока в сети. Как следствие описанный выше алгоритм следует применять на практике только тогда, когда пропускные способности ребер не велики. Примером сети, на которой обобщенный алгоритм дополняющего пути будет работать чрезмерно долго, является сеть, приведенная на рис. 8, с пропускными способностями ребер установленными в $1{,}000{,}000{,}000$. Следует отметить, что алгоритм будет работать долго тогда и только тогда, ко-

гда в качестве дополняющего пути будет выбираться либо путь $s \rightsquigarrow 1 \rightsquigarrow 2 \rightsquigarrow t$, либо путь $s \rightsquigarrow 2 \rightsquigarrow 1 \rightsquigarrow t$ (см. рис. 8).

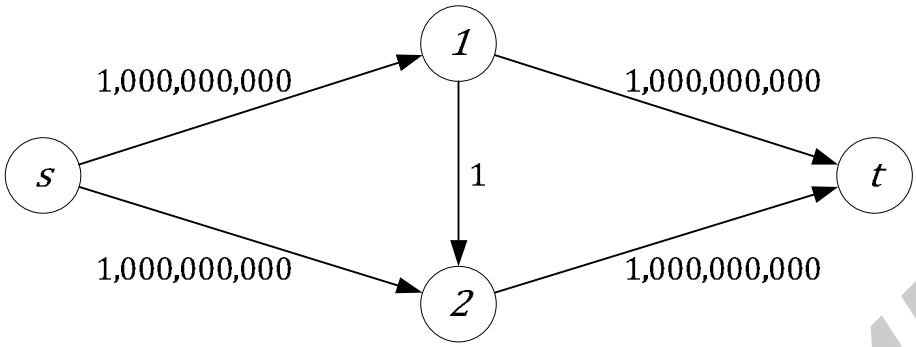


Рис. 8. «Наихудшая» сеть для алгоритма дополняющего пути.

3.3. Алгоритм пометок

В этом разделе рассматривается реализация алгоритма дополняющего пути, предложенная Эдмондсом и Карпом в 1972 году [8]. Суть метода заключается в поиске кратчайшего дополняющего пути с точки зрения количества использованных дуг. Следовательно, изменений требует лишь функция *augment()*. Код функции приведен в листинге 12.

Листинг 12. Функция augment().

```
1.  bool augment(tgraph &g, int nvertices, int source, int sink)
2.  {
3.      for (int i = 1; i <= nvertices; ++i)
4.          parent[i] = 0;
5.      queue<int> q;
6.      for (q.push(source); !q.empty(); q.pop())
7.      {
8.          int v = q.front();
9.          for (tedge* x = g[v]; x != 0; x = x->next)
10.             if (parent[x->destination] == 0 &&
11.                 x->capacity - x->flow > 0)
12.             {
13.                 parent[x->destination] = x;
14.                 q.push(x->destination);
15.             }
16.     }
17.     return (parent[sink] != 0);
18. }
```

В [8] показано, что алгоритмическая сложность алгоритма Эдмондса-Карпа равняется $O(|V|*|E|^2)$, где $|V|$ – количество вершин в сети, $|E|$ – количество ребер в сети. Как следствие описанный выше алгоритм следует применять на практике для поиска максимального потока в сети с небольшим количеством вершин и ребер.

3.4. Алгоритм Диница

Перед тем, как перейти к непосредственному рассмотрению алгоритма Диница [9], введем ряд необходимых определений.

Рассмотрим сеть $G=(V,E)$ с истоком s и стоком t. Пусть на этой сети задана функция пропускной способности c и некоторый поток f. Остаточную сеть обозначим через G_f.

Определение 22. *Функцией расстояния* для остаточной сети G_f назовем такую функцию $d: V \to \boldsymbol{Z}^+ \cup \{0\}$, для которой выполнены следующие условия

- $d(s)=0$;
- $d(v) \le d(u)+1, \forall (u,v) \in G_f$.

В этом случае $d(u)$ есть расстояние от истока s до вершины v.

Далее предположим, что для остаточной сети $G_f=(V,E_f)$ задана функция расстояния d [9,10].

Определение 23. Дуга (u,v) сети G_f называется *допустимой*, если выполнено условие

$$d(v)=d(u)+1.$$

Все остальные дуги назовем *недопустимыми*.

Определение 24. Назовем простой путь из истока s в сток t в сети G_f *допустимым*, если все дуги этого пути являются допустимыми.

Ключевое свойство допустимых путей, устанавливающее связь с дополняющими путями, сформулировано в следующей лемме.

Лемма 1. Допустимый путь является кратчайшим дополняющим путем из истока s в сток t в сети G_f.

На рис. 9 приведен пример функции расстояния для сети рис. 6.

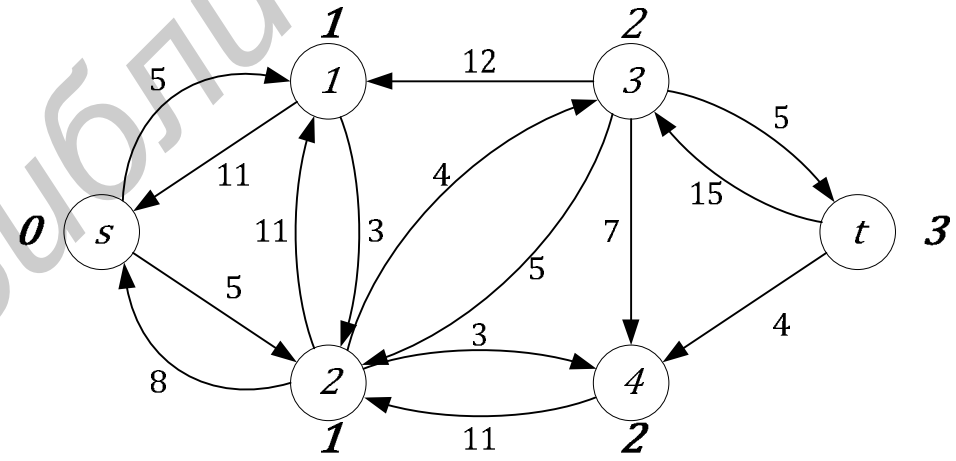


Рис. 9. Пример функции расстояния d.
Рядом с вершиной u указано значение $d(u)$.

Определение 25. *Допустимой сетью* для сети G_f назовем такую подсеть L_f, дуги которой являются допустимыми.

На рис. 10 приведен пример допустимой сети для сети рис. 9.

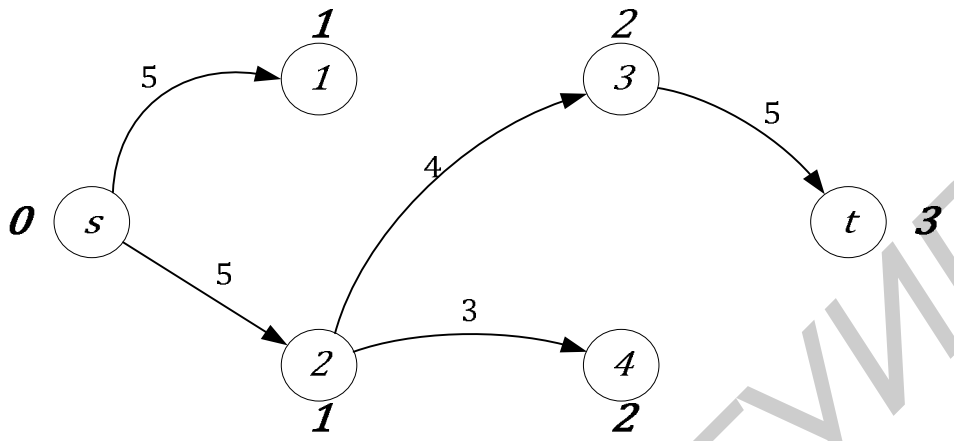


Рис. 10. Допустимая сеть L_f для сети рис. 9.

Функция расстояния и допустимая сеть для заданной остаточной сети, могут быть построены прямым применением поиска в ширину, листинг 13[4].

Листинг 13. Построение функции расстояния и допустимой сети.

```
1.  bool bfs(tgraph &g, int nvertices, int source, int sink)
2.  {
3.      for (int i = 1; i <= nvertices; ++i)
4.          distance[i] = undefined;
5.      distance[source] = 0;
6.      queue<int> q;
7.      for (q.push(source); !q.empty(); q.pop())
8.      {
9.          int v = q.front();
10.         for (tedge* x = g[v]; x != 0; x = x->next)
11.         if (distance[x->destination] == undefined &&
12.             x->capacity - x->flow > 0)
13.         {
14.             distance[x->destination] = distance[v] + 1;
15.             q.push(x->destination);
16.         }
17.     }
18.     return (distance[sink] != undefined);
19. }
```

[4] Следует обратить внимание на тесную связь данного алгоритма с алгоритмами поиска дополняющего пути, приведенных в листингах 10 и 12. Как следствие, данный алгоритм может быть применен для проверки того, существует ли дополняющий путь в остаточной сети.

Как и в алгоритмах, рассмотренных ранее, процедуре *bfs()* анализирует только те ребера, чья остаточная пропускная способность строго положительна (строка 12). Вычисление же расстояния до вершины происходит только один раз (строка 14) и осуществляется только тогда, когда расстояние до вершины в текущий момент неизвестно (строка 11). Достичь такого поведения можно путем инициализации массива *distance* значением *undefined* (строки 3-4) перед началом выполнения основного цикла (строки 7-17).

Определение 26. Дуга (u,v) называется *насыщенной*, если величина потока $f(u,v)$ по дуге (u,v) совпадает с пропускной способностью дуги $c(u,v)$

Определение 27. *Блокирующим потоком* для сети G с истоком s и стоком t называется такой поток f, для которого любой путь из s в t содержит насыщенную дугу.

Следует отметить, что блокирующий поток необязательно является максимальным потоком. Пример блокирующего потока приведен на рис. 11.

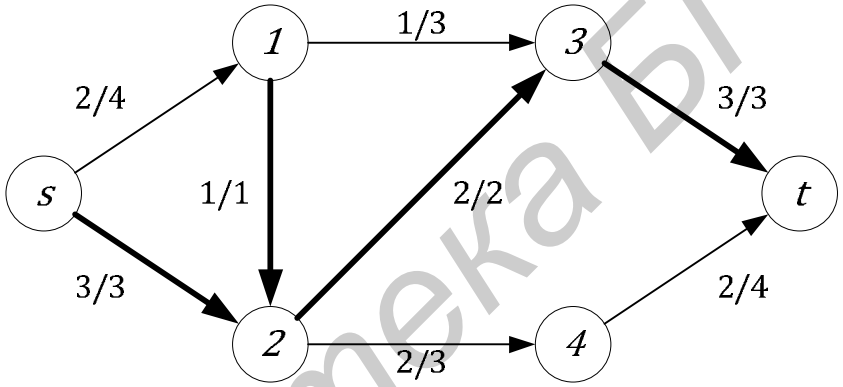


Рис. 11. Пример блокирующего потока величины 5.
Максимальный же поток равен 6.

Идея алгоритма Диница заключается в нахождении блокирующего потока в допустимой сети [9]. Процедура поиска блокирующего потока может быть реализована с применением поиска в глубину, листинг 14.

Листинг 14. Построение блокирующего потока.

```
1.  void build_blocking_flow(tgraph &g, int source, int sink)
2.  {
3.      while (dfs(g, source, 0, sink))
4.      {
5.          tpath path = get_augmenting_path(source, sink);
6.          int delta = +oo;
7.          for (size_t i = 0; i < path.size(); ++i)
8.          {
9.              tedge *x = path[i];
10.             delta = min(delta, x->capacity - x->flow);
11.         }
```

```
12.         for (size_t i = 0; i < path.size(); ++i)
13.         {
14.             tedge *x = path[i];
15.             x->flow += delta;
16.             if (x->reverse != 0)
17.                 x->reverse->flow -= delta;
18.         }
19.     }
20. }
```

Обратим внимание, что тело функции *build_blocking_flow()* практически совпадает с телом функции *generic_augmenting_path_algorithm* (листинг 8), с тем лишь различием, что функция *build_blocking_flow()* пропускает поток вдоль всех допустимых путей (а значит и дополняющих путей по лемме1). Функция поиска допустимого пути будет практически совпадать с функцией поиска дополняющего пути листинга 10, с тем лишь различием, что анализироваться будут только допустимые дуги (листинг 15, строка 9).

Листинг 15. Поиск допустимого пути.

```
1.  bool dfs(tgraph &g, int source, tedge *by_edge, int sink)
2.  {
3.      parent[source] = by_edge;
4.      if (source == sink)
5.          return true;
6.      for (; g[source] != 0; g[source] = g[source]->next)
7.      {
8.          tedge *x = g[source];
9.          if (distance[x->destination] == distance[source] + 1 &&
10.             x->capacity - x->flow > 0)
11.         {
12.             bool result = dfs(g, x->destination , x, sink);
13.             if (result)
14.                 return true;
15.         }
16.     }
17.     return false;
18. }
```

Поясним вкратце принцип работы процедуры *build_blocking_flow()*. Как и в алгоритме дополняющего пути (листинг 10), запускается поиск в глубину, просматривающий допустимые ребра. После того, как достигнут сток *sink* , процедура *dfs()* немедленно возвращается с положительным результатом. Далее процедура *build_blocking_flow()* пропускает поток вдоль пути, найденного функцией *get_augmenting_path()*.

С учетом рассмотренных выше процедур, можно предложить следующую реализацию алгоритма Диница, приведенную в листинге 16.

Листинг 16. Алгоритм Диница.

```
1.  int dinic_algorithm(tgraph &g, int nvertices, int source, int sink)
2.  {
3.      while (bfs(g, nvertices, source, sink))
4.      {
5.          for (int i = 1; i <= nvertices; ++i)
6.              _g[i] = g[i];
7.
8.          build_blocking_flow(_g, source, sink);
9.      }
10.     return get_flow(g, source, sink);
11. }
```

В [9] показано, что алгоритмическая сложность данного алгоритма равняется $O(|V|^2 * |E|)$, где $|V|$ – количество вершин в сети, $|E|$ – количество ребер в сети. В то же самое время, существуют структуры данных, такие как динамические деревья [1,5], позволяющие уменьшить время выполнения алгоритма Диница до $O(|E| * \log(|V|))$. Несмотря на это, базовая реализация алгоритма Диница показывает неплохие результаты на практике и может быть использована для нахождения потока в сетях с количеством вершин до $5{,}000$ и количеством ребер до $30{,}000$ включительно.

3.5. Заключение

В заключение данного раздела дадим краткий обзор задач теории графов и, в целом, задач комбинаторики, которые эффективно решаются путем применения алгоритмов нахождения максимального потока в сети.

Прежде всего, коснемся вопросов, связанных с топологией самих сетей. Пусть нам задана сеть $G = (V, E)$ с истоком s и стоком t

Определение 28. Два пути из истока s в сток t в сети $G = (V, E)$ называются *реберно-независимыми*, если они не имеют общих ребер.

Определение 29. Два пути из истока s в сток t в сети $G = (V, E)$ называются *вершинно-независимыми*, если они не имеют общих вершин.

Ниже приводится ряд теорем, устанавливающих некоторые из базовых свойств реберно-независимых и вершинно-независимых путей.

Теорема 3. Максимальное количество реберно-независимых путей из истока s в сток t в сети $G = (V, E)$ равняется минимальному числу ребер, удаление которых приводит к разрыву всех путей из истока s в сток t.

Теорема 4. Максимальное количество вершинно-независимых путей из истока s в сток t в сети $G = (V, E)$ равняется минимальному числу вершин, удаление которых приводит к разрыву всех путей из истока s в сток t.

Доказательство данных теорем опирается на понятие разреза в сети. Сами же теоремы были впервые сформулированы ученым Менгером в 1927 году [1].

Определение 30. *Числом реберной связности* $k_e(G)$ графа G называется минимальное количество ребер, удаление которых приводит к разделению графа на несколько компонент связности.

С другой стороны, под числом реберной связности графа G можно понимать величину минимального разреза в G. Для того, чтобы дать конструктивный алгоритм для определения числа $k_e(G)$, введем следующее определение

Определение 31. *Локальным числом реберной связности* $k_e(s,t)$ между вершинами s и t графа G называется минимальное количество ребер, удаление которых приводит к разрыву всех путей из вершины s в вершину t.

С учетом определения 30 для числа $k_e(G)$ получим

$$k_e(G) = \min_{s,t \in V} \{k_e(s,t)\}.$$

Алгоритм нахождения числа $k_e(G)$ приведен в листинге 17.

Листинг 17. Алгоритм нахождения числа реберной связности.

```
1.  int get_edge_connectivity_number(tgraph &g)
2.  {
3.      build_unit_capacitated_network(network, g);
4.      int source = 1, sink;
5.      int Ke = |E|;
6.      for (sink = 2; sink <= |V|; ++sink)
7.      {
8.          int flow = get_max_flow(network, |V|, source, sink);
9.          Ke = min(Ke, flow);
10.     }
11.     return Ke;
12. }
```

Обратим внимание на строку 3, в которой происходит построение сети для входного графа. Пропускная способность на дугах такой сети устанавливается в 1. Далее перебирается исток *sink* (строка 6) и находится максимальный поток в сети *network* (строка 8). Алгоритмическая сложность полученного алгоритма равняется $O(|V|*MF(|V|,|E|,C))$, где $MF(|V|,|E|,C))$ – время, требуемое для нахождения максимального потока в сети с $|V|$ вершинами, $|E|$ ребрами и максимальной пропускной способностью ребер равной C.

Определения 30, 31 и алгоритм из листинга 17 явным образом формулируются и для вершин, с некоторыми лишь изменениями.

Еще одним примером применения теории потоков к задачам комбинаторики является задача нахождения максимального паросочетания в двудольных графах. Многие из алгоритмов нахождения максимального потока были адаптированы специально для этих целей [1,5].

Нельзя не отметить тот факт, что в пособии были рассмотрены далеко не все техники, которые позволяют эффективно решать задачу о поиске наибольшего потока в сети с ограничениями на пропускные способности ребер. Одной

из них является техника «проталкивания предпотока». Данная теория получила широкое распространение и, как следствие, одни из самых эффективных алгоритмов базируются именно на ней [1,6]. Что же касается рассмотренных выше алгоритмов, то они являются базовыми и любое изучение теории потоков стоит начинать именно с них, так как они послужили основным толчком к дальнейшим исследованиям. Большинство же задач, которые предлагаются на соревнованиях по программированию, имеют под собою решения, которые можно эффективно реализовать, опираясь на рассмотренные выше подходы.

3.6. Примеры решения задач

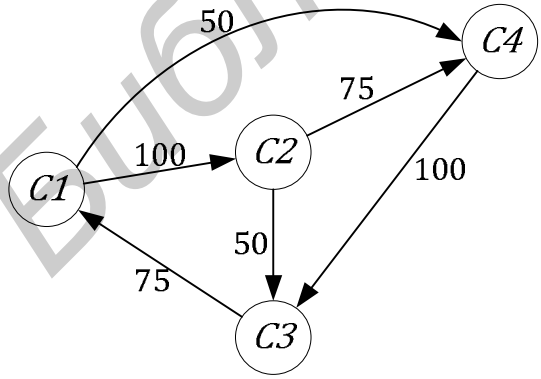
Задача 1. Из-за экономического кризиса многие предприятия, являющиеся клиентами банка, не могут получить долги от покупателей и рассчитаться с продавцами. Банк хранит информацию о долгах клиентов в виде долговых обязательств. Долговое обязательство представляет собой тройку натуральных чисел: номер должника, номер предприятия, которому он должен, и сумму долга. Банк намерен выполнить взаимозачет долгов своих клиентов. Для этого банк может изменять долговые обязательства своих клиентов любым образом при условии, что для каждого клиента останется неизменным сальдо.

Требуется преобразовать заданный список долговых обязательств в иной список, в котором общая сумма долгов минимальна.

Решение. Прежде всего, построим математическую модель задачи. Для этого занумеруем всех клиентов банка от 1 до n. Обозначим через $credit(i)$ – сумму всех кредитов, которые закреплены за клиентом i. Обозначим через $debt(i)$ – сумму всех долгов, которые закреплены за клиентом i. Тогда сальдо $balance(i)$ клиента i определяется как:

$$balance(i) = credit(i) - debt(i).$$

Для того чтобы дать введенным определениям большую наглядность, приведем пример (рис. 12).



а)

#	Credit	Debt	Balance
C1	75	150	-75
C2	100	125	-25
C3	150	75	75
C4	125	100	25

б)

Рис. 12. а) Долговые обязательства клиентов, представленные в виде графа;
б) суммарный кредит, долг и сальдо для каждого из клиентов.

Будем говорить, что набор долговых обязательств является *сбалансированным,* если сумма сальдо всех клиентов равняется 0. Набор долговых обязательств, приведенный на рис. 12, является сбалансированным.

Условие сбалансированности может быть пояснено следующим образом. Действительно, если клиент $C1$ должен клиенту $C2$ сумму денег x, то сальдо клиента $C1$ уменьшится на x, а сальдо клиента $C2$ увеличится на x. Если взять такие суммы по всем долговым обязательствам, то в результате получится 0. С точки зрения банка это означает, что денежные потоки осуществляются только между клиентами этого банка.

Следующим шагом будет разделение всех клиентов банка на 3 группы:

- *Кредиторы (Creditors).* Клиенты, сальдо которых строго положительно.
- *Должники (Debtors).* Клиенты, сальдо которых строго отрицательно.
- *Расплатившиеся.* Клиенты, сальдо которых равняется 0.

В дальнейшем будем рассматривать только кредиторов и должников, т.к. для выполнения взаимозачета долгов банку незачем рассматривать клиентов, которые не участвуют в денежных потоках. Также отметим, что клиент не может принадлежать более чем 1 группе.

Далее введем понятия *центра кредитов (ЦК)*, который выдает деньги клиентам, и *центра сборов (ЦС)*, который собирает с клиентов долги. С учетом введенных понятий, пример рис. 12(а) можно преобразовать в граф, изображенный на рис. 13.

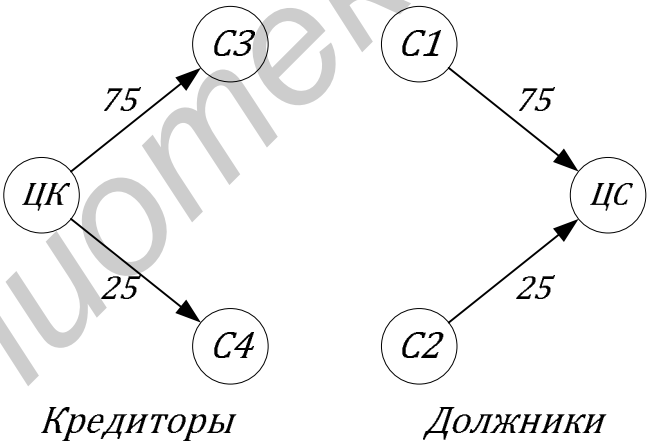


Рис. 13. Преобразованный граф рис. 12.

Поясним смысл введенных терминов. Центр кредитов является тем подразделением банка, которое занимается выдачей денег клиентам. Клиенты же в свою очередь могут распоряжаться полученными деньгами как угодно. Центр сборов отслеживает суммы, которые клиенты могли получить от других клиентов того же банка. Стало быть, решение поставленной задачи – это некоторый набор кредитов из группы «кредиторы» в группу «должники», причем сумма этих кредитов должна быть минимальной.

Обозначим искомую сумму новых кредитов через *totalCredit*. Далее заметим, что в силу сбалансированности долговых обязательств, можно заключить, что общая сумма всех кредитов *totalCredit* не может превышать общего кредита от ЦК «кредиторам» (так как деньги не могут взяться из «ниоткуда»). В то же самое время, *totalCredit* не может быть меньше суммы всех долгов (так как деньги не могут уходить в «никуда»). В силу сбалансированности долговых обязательств можно заключить, что для величины *totalCredit* справедливы следующие равенства

$$totalCredit = \sum_{i \in Creditors} balance(i) = \sum_{i \in Debtors} |balance(i)|.$$

Все что осталось сделать – распределить деньги от кредиторов должникам. Для этого можно построить сеть $G = (V, E)$ аналогичную сети, приведенной на рис. 14.

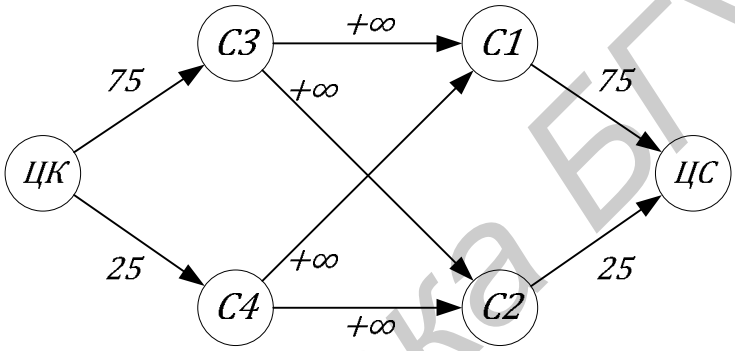


Рис. 14. Сеть для графа рис. 13.

Отметим, что пропускная способность ребер из группы «кредиторы» в группу «должники» установлена в $+\infty$. Сделано это потому, что величина минимального разреза данной сети должна быть равна *totalCredit*. Теорема Форда-Фалкерсона [7] дает способ построения новых долговых обязательств: для этого достаточно построить максимальный поток в сети. Численное значение долговых обязательств будет совпадать с величиной потока по ребрам из группы «кредиторы» в группу «должники». Пример приведен на рис. 15.

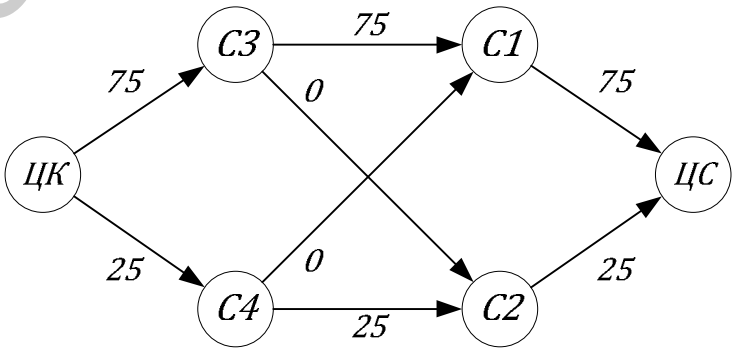


Рис. 15. Найденные значения долговых обязательств.

Задача 2. Рассмотрим ориентированный граф $O=(V,E)$. *Степенью ориентированности* графа O назовем число $D(O)$, равное максимуму из всех чисел $\deg^-(v)$, $v\in V$. Требуется так ориентировать заданный неориентированный граф $G=(V,E)$, чтобы степень его ориентированности была минимальной.

Решение. Перед тем как искать решение, приведем пример (рис. 16). Будем говорить, что граф $G=(V,E)$ *ориентирован числом* d, если $D(G)=d$. Тогда очевидным подходом к решению данной задачи является перебор искомой степени ориентированности d и проверки того, можно ли заданный граф ориентировать так, чтобы степень $D(G)$ была равна d.

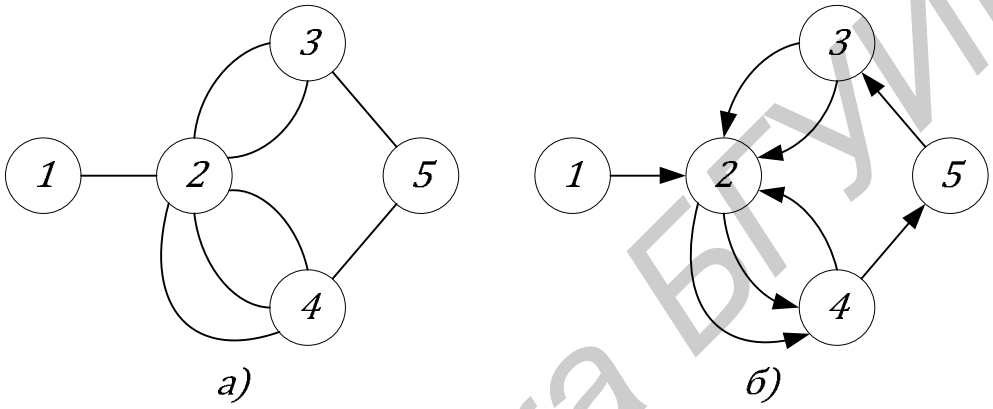


Рис. 16. а) Исходный граф $G=(V,E)$;
б) полученный ориентированный граф $G*$, для которого $D(G*)=2$.

Листинг 18. Поиск степени ориентированности.

```
1. for (int d = 1; d <= |E|; ++d)
2. {
3.     if (direct(graph, d))
4.     {
5.         return d;
6.     }
7. }
```

Попробуем проанализировать граф G, с целью обнаружения таких его свойств, которые помогут построить функцию *direct()* (строка 3), работающую за полиномиальное время. Для этого рассмотрим пару вершин u и v графа G. Пусть эти вершины соединяет в точности k ребер. Так как после преобразования ребра станут ориентированными, то можно будет точно сказать, сколько дуг идет из u в v, а сколько – из v в u. Пусть эти числа равны k_1 и k_2 соответственно. Тогда группу ребер, идущих из u в v заменим единственной дугой с весом $c(u,v)=k_1$, группу ребер, идущих из v в u – дугой с весом $c(v,u)=k_2$. На рис. 17 приведена последовательность описанных выше преобразований.

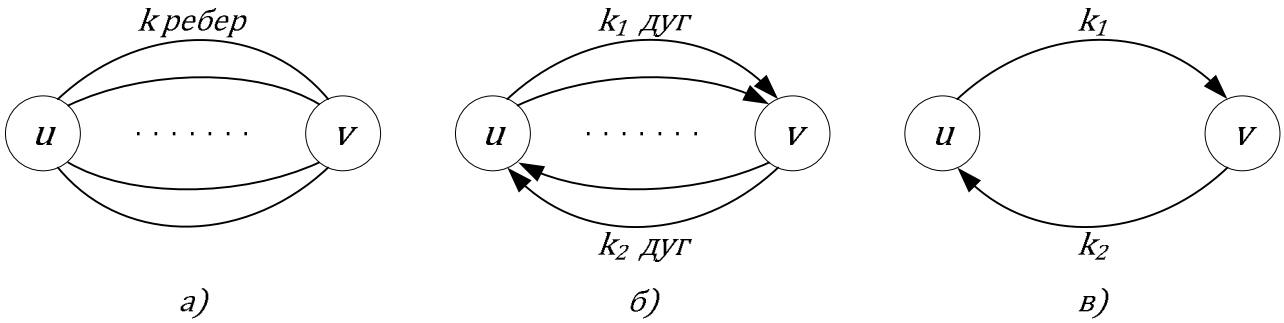


Рис. 17. Преобразование мультиребра к паре дуг.

Последнее преобразование для данной пары вершин будет заключаться в ведении дополнительной вершины uv, рис. 18.

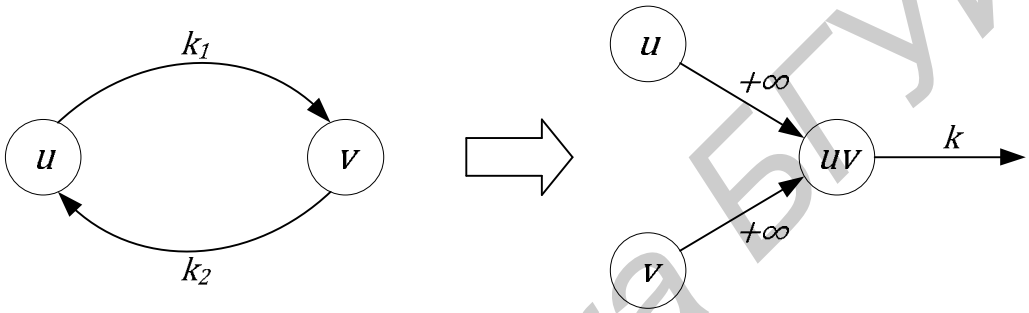


Рис. 18. Введение дополнительной вершины.

Суть этого преобразования состоит в том, что перед началом выполнения алгоритма не известно, какое количество ребер будет направлено из u в v, а какое – из v в u. Известна же только сумма числе k_1 и k_2, равная k.

Следующим шагом будет преобразование исходного графа в сеть, с учетом описанных выше соображений. На рис.19(б) приведен пример такой сети для графа рис. 16. Данная сеть строится по всем парам вершин, которые соединены ребрами. Следует обратить внимание на пропускную способность дуг, ведущих из истока s в вершины графа, равную значению d, где d – перебираемая оценка искомой степени ориентированности. Применение к построенной сети одного из алгоритмов нахождения максимального потока, позволяет определить, можно ли ориентировать граф так, чтобы его степень ориентированности была равна d. Это возможно тогда и только тогда, когда величина максимального потока будет равна $|E|$, где $|E|$ – число ребер заданного графа. Это вытекает из того, что запущенный поток должен насытить все ребра, ведущие из дополнительных вершин в исток t.

Собрав все вместе, можно предложить следующую реализацию функции поиска минимальной ориентации заданного графа G, листинг 19.

Листинг 19. Функция direct().

```
1. bool direct(tgraph &g, int d)
2. {
3.     tgraph network = build_network_by(g);
4.     return (get_max_flow(network, d, source, sink) == |E|);
5. }
```

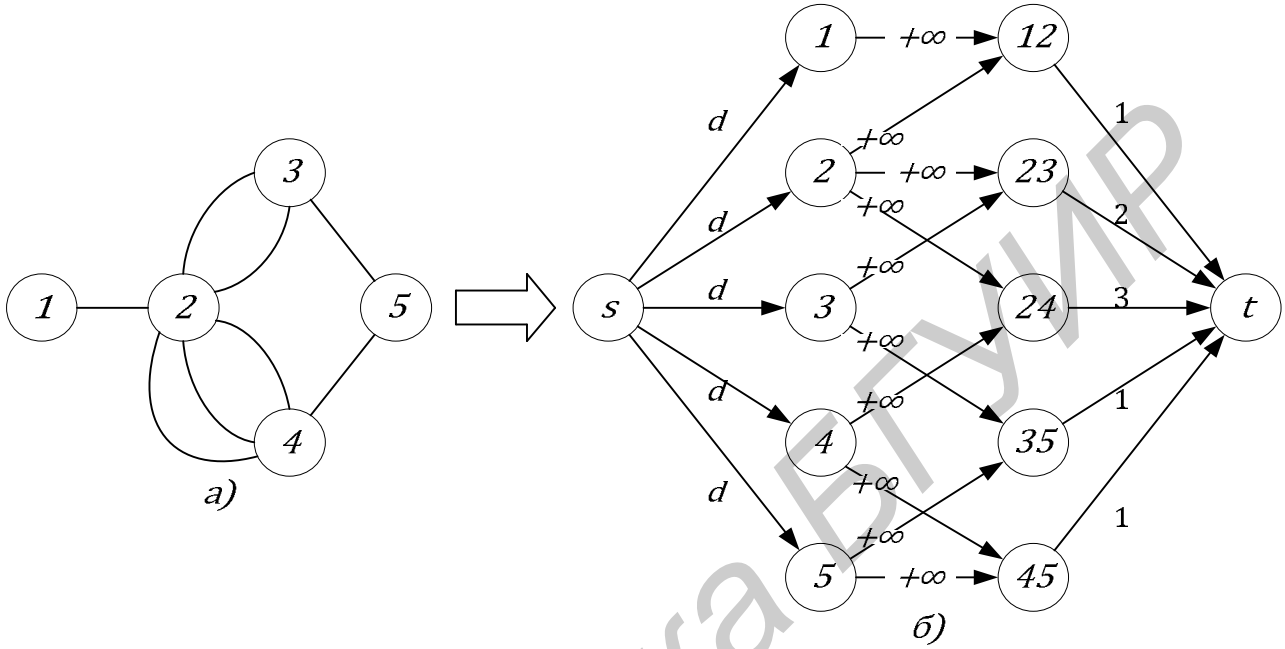


Рис. 19. а) Исходный граф $G = (V, E)$; б) Сеть для графа G.

В заключение разбора данной задачи заметим, что вместо линейного поиска, используемого в листинге 18, можно применить двоичный. Вытекает это из того, что построение максимального потока величины $|E|$ для оценки d, влечет за собою существование такого же потока и для оценки $d' > d$, в силу бесконечных пропускных способностей дуг, ведущих из оригинальных вершин графа в дополнительные (см. рис 19(б)). Пример кода приведен в листинге 20.

Листинг 20. Поиск степени ориентированности.

```
1.  tgraph network = build_network_by(g);
2.  int lo = 0, hi = |E|;
3.  while (lo + 1 < hi)
4.  {
5.      int d = (lo + hi) / 2;
6.      if (get_max_flow(network, d, source, sink) == |E|)
7.          hi = d;
8.      else
9 .         lo = d + 1;
10. }
11. int result =
12.     (get_max_flow(network, lo, source, sink) == |E| ? lo : hi);
13. return result;
```

Важным является то, что построение сети *network* происходит всего лишь один раз (строка 1). Функция *get_max_flow()* в начале своей работы устанавливает пропускные способности дуг, выходящих из истока s, в значение d и обнуляет поток на всех ребрах сети. В заключение отметим, что использование алгоритма Диница в качестве реализации функции *get_max_flow()* позволяет решать данную задачу с количеством вершин и ребер до $10,000$ включительно.

3.7. Задачи для самостоятельного решения

Задача 3. Судоходная компания заключила контракт на поставку скоропортящихся грузов между различными портами. Поскольку грузы являются скоропортящимися, клиенты установили точные даты их поставки и прибытия, т.е. даты отправления грузов из пунктов производства и прибытия в пункты назначения (грузы не могут быть доставлены раньше или позже). Судоходная компания хотела бы определить минимальное количество кораблей, необходимое для доставки грузов в срок.

В качестве иллюстрации к данной задаче, приведем пример. Рассмотрим 4 поставки, характеристики которых приведены на рис. 20.

Поставки

#	Откуда	Куда	Дата
1	Port A	Port C	3
2	Port A	Port C	8
3	Port B	Port D	3
4	Port B	Port C	6

а)

Время перевозки

#	A	B	C	D
A	0	1	3	2
B	1	0	2	3
C	3	2	0	4
D	2	3	4	0

б)

Время возвращения

#	A	B	C	D
A	0	4	2	1
B	4	0	1	2
C	2	1	0	3
D	1	2	3	0

в)

Рис. 20. а) План поставок; б) время, требуемое для перевозки грузов; в) время, требуемое для возвращения.

Минимальное количество кораблей, которое потребуется для осуществления поставок, приведенных на рис. 20, равняется 2.

Задача 4. Рассмотрим игру, состоящую из маленьких платформ и труб. Платформы разделяются на стартовые (N_1 штук), промежуточные (N_2 штук) и финишные (N_3 штук). Каждой платформе соответствует уникальный номер от 1 до $N_1 + N_2 + N_3$. Нумерация платформ начинается со стартовых, затем идут промежуточные и, наконец, финишные. Все промежуточные платформы пронумерованы по убыванию высоты. Стартовые платформы находятся на высоте H_s, финишные – на высоте H_f. Высоты $H_m(i)$ промежуточных платформ различны ($H_f < H_m(i) < H_s$,). На каждой из стартовых платформ находится шарик.

Шарик может скатиться с платформы A на платформу B, если они соединены трубой, и высота A больше высоты B. На каждой из финишных платформ может оказаться не более одного шарика. Для каждой промежуточной платформы задано число K, равное максимальному количеству шариков, которые могут прокатиться по ней за время игры. Вам требуется узнать, какое максимальное количество шариков может оказаться на финишных платформах в конце игры.

Задача 5. Рассмотрим $0-1$ матрицу A размерности $n \times n$. Обозначим через α вектор, i-ый элемент которого равен сумме элементов i-ой строки матрицы A. Обозначим через β вектор, i-ый элемент которого равен сумме элементов i-ого столбца матрицы A. Будем считать, что векторы α и β упорядочены по убыванию. Формально это выглядит так

$$\alpha_i = \sum_{j=1}^{n} A[i,j],\ \forall i = \overline{1,n};\ \beta_i = \sum_{j=1}^{n} A[j,i],\ \forall i = \overline{1,n}.$$

$$\alpha_1 \geq \alpha_2 \geq \ldots \geq \alpha_n;\ \beta_1 \geq \beta_2 \geq \ldots \geq \beta_n.$$

Вам заданы векторы α и β размерности n каждый. Предложите способ, который позволит по данным векторам восстановить матрицу A, либо сообщит, что такой матрицы не существует. Если же матрица A не единственна, то достаточно построить какую-нибудь одну.

Задача 6. Пусть задана сеть $G = (V, E)$ с функцией пропускной способности U. Предположим, что помимо верхней границы на пропускные способности дуг задана и нижняя граница, т.е. такая функция L, для которой выполняются следующие условия

- $L(u,v) \geq 0,\ \forall (u,v) \in E$;
- $L(u,v) = 0,\ \forall (u,v) \notin E$.

Тогда на функцию потока f накладывается ограничение вида

$$L(u,v) \leq f(u,v) \leq U(u,v).$$

Вам предлагается модифицировать рассмотренные выше алгоритмы так, чтобы они позволяли находить максимальный поток в сети с двусторонними ограничениями. В то же самое время вы можете изменить входную сеть таким образом, что бы базовые алгоритмы смогли подойти для этих целей. Следует отметить, что поток в сетях такого типа может и не существовать. Пример такой сети приведен на рис. 21.

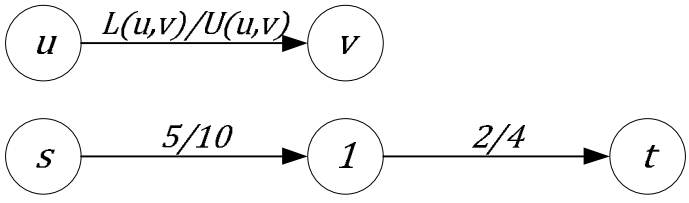


Рис. 21. Пример сети, в которой не существует потока.

Задача 7. Рассмотрим сеть $G=(V,E)$ с заданной функцией пропускной способности c. Назовем дугу (u,v) сети G *критической сверху*, если увеличение пропускной способности $c(u,v)$ дуги (u,v) приводит к увеличению величины максимального потока. Предложите алгоритм, который найдет все критические сверху дуги заданной сети G. Обратим внимание на то, что время работы построенного алгоритма не должно превышать время работы алгоритма нахождения максимального потока. Проведите те же исследования, но уже для *критических снизу* дуг, т.е. таких дуг, уменьшение пропускной способности которых влечет уменьшение величины максимального потока. Всегда ли множества критических сверху и снизу дуг совпадают?

4. Паросочетания в двудольных графах

Задача поиска паросочетаний в графах возникает на практике довольно часто. Причем интерес представляет поиск наибольшего паросочетания [1,10]. Неформально постановка задачи может быть сформулирована следующим образом: «Задан перечень пар («мальчик», «девочка»), которые согласны танцевать друг с другом в паре. Какое максимальное количество пар сможет выступить на концерте?».

Перед тем как перейти к непосредственному рассмотрению методов решения таких типов задач, введем ряд важных определений.

4.1. Основные понятия

Определение 32. Пусть задан двудольный граф $G = (V_1 \cup V_2, E)$. *Паросочетанием* называется такое множество ребер $M \subseteq E$, в котором никакие два ребра не являются инцидентными.

Пример паросочетания приведен на рис. 22.

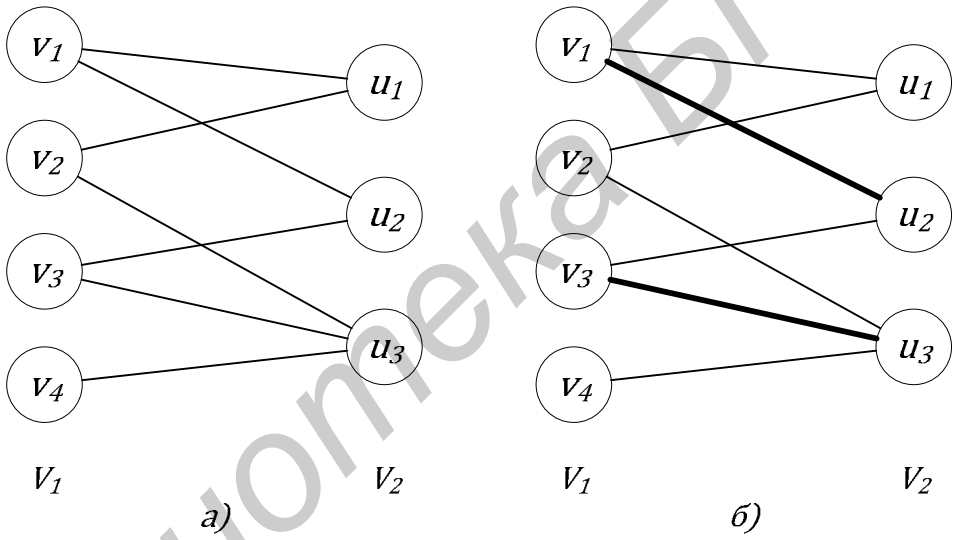


Рис. 22. а) Пример двудольного графа $G = (V_1 \cup V_2, E)$ рис. 1.
б) Паросочетание $M = \{(v_1, u_2),\ (v_3, u_3)\}$

Определение 33. Паросочетание M называется *максимальным*, если оно содержит максимальное количество ребер, т.е. не существует такого паросочетания M', для которого $|M'| > |M|$.

Здесь следует отметить тесную связь задачи поиска наибольшего паросочетания с задачей нахождения максимального потока в сети [1,5]. Для этого к рассматриваемому двудольному графу добавим две вершины, одна из которых исток s, а вторая сток t. Соединим исток ребрами с вершинами первой доли. Соединим сток ребрами с вершинами второй доли. Пропускную способность ребер построенной сети установим в 1. Применение какого-нибудь из алгорит-

мов построения максимального потока позволяет построить и максимальное паросочетание. Сеть, построенная для графа рис. 22, приведена на рис. 23.

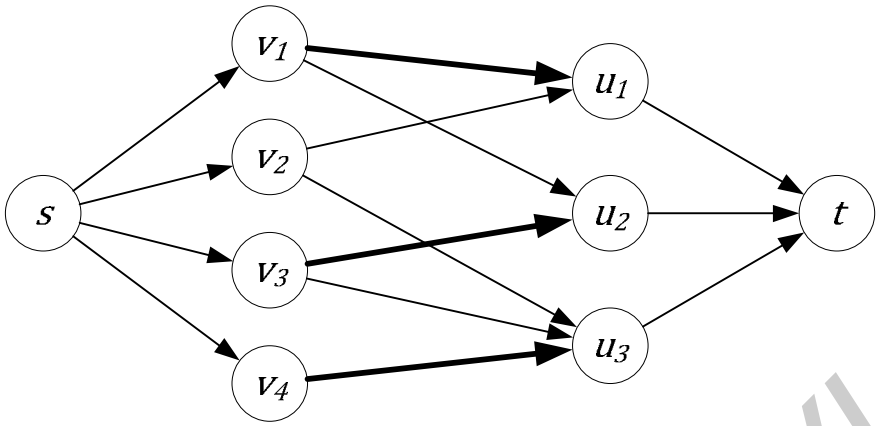


Рис. 23. Максимальный поток в сети, построенной для графа рис. 22.

В силу специфики рассматриваемой задачи, на практике не рекомендуется использовать алгоритмы нахождения максимального потока. С учетом же особенностей данной задачи учеными были разработаны более эффективные методики, позволяющие решить задачу поиска максимального паросочетания в двудольном графе за разумное время.

Далее рассмотрим граф $G = (V_1 \cup V_2, E)$, для которого построено некоторое, необязательно максимальное, паросочетание M.

Определение 34. Вершина v называется *свободной* относительно паросочетания M, если не существует ребра $(v,u) \in M$.

В примере рис. 22(б) вершины v_4 и u_1 являются свободными.

Определение 35. *Чередующейся цепью* относительно паросочетания M назовем такую цепь, в которой ребра поочередно принадлежат и не принадлежат паросочетанию.

Пример чередующейся цепи для графа рис. 22(б) приведен на рис. 24.

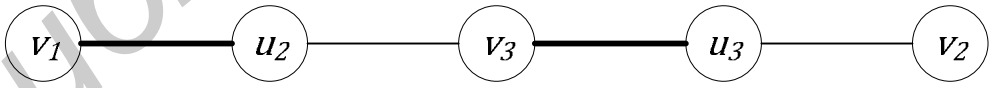


Рис. 24. Пример чередующейся цепи.

Определение 35. *Дополняющей цепью* относительно паросочетания M назовем чередующуюся цепь, в которой начальная и конечная вершины являются свободными.

Пример дополняющей цепи для графа рис. 22(б) приведен на рис. 25.

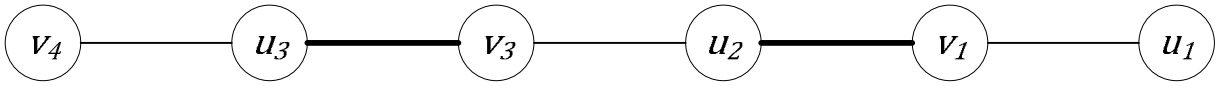


Рис. 25. Пример дополняющей цепи.

Заметим, что количество ребер в дополняющей цепи нечетно и крайние вершины цепи принадлежат различным долям графа. Как следствие, замена паросочетания M паросочетанием M', в которое будут входить ребра, не входящие в паросочетание M, приведет к увеличению паросочетания на 1 [1,6,10]. Пример данного преобразования для цепи рис. 25 приведен на рис. 26.

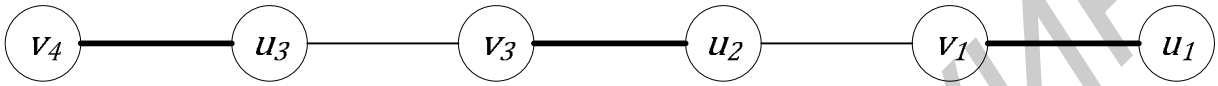


Рис. 26. Увеличение мощности паросочетания на 1.

Следует отметить, что все современные методы нахождения максимального паросочетания в графе базируются на теории дополняющих путей и теореме Бержа, сформулированной и доказанной Бержем в 1957 году[11].

Теорема 3. Паросочетание M в двудольном графе $G = (V_1 \cup V_2, E)$ является максимальным тогда и только тогда, когда в графе G не существует дополняющей цепи относительно паросочетания M.

4.2. Алгоритм Куна

Идея алгоритма Куна базируется на теореме 3 [12]. Применив подход, аналогичный тому, который применялся для алгоритмов нахождения максимального потока, можно построить следующую функцию нахождения максимального паросочетания, листинг 21.

Листинг 21. Алгоритм Куна[5].

```
1.  int kuhn(tgraph &g, int npart1, int npart2)
2.  {
3.      int cardinality = 0;
4.      for (int i = 1; i <= npart1 + npart2; ++i)
5.          matching[i] = undefined;
6.      while (augment(g, npart1, npart2))
7.      {
8.          cardinality = cardinality + 1;
9.      }
10.     return cardinality;
11. }
```

[5] Будем считать, что вершины графа занумерованы от 1 до *npart1* + *npart2*, где *npart1* – количество вершин в первой доле, *npart2* – количество вершин во второй доле.

Рассмотрим массив *matching*, в котором будем хранить текущее найденное паросочетание. Елемент i массива *matching* указывает на то, с какой вершиной будет связана вершина i в максимальном паросочетании. В начале паросочетание пусто, поэтому в строках 4-5 происходит инициализация массива *matching* значением *undefined* равным, например, -1. По завершении цикла в строках 6-9 элементы массива *matching*, отличные от *undefined*, могут быть использованы для формирования паросочетания в виде ребер $(i, matching[i])$.

Как и в общем алгоритме дополняющего пути (листинг 10), функция *augment()* (строка 6) может быть реализована с использованием поиска в глубину. Пример реализации приведен в листинг 22.

Листинг 22. Функция augment().

```
1.  bool augment(tgraph g, int npart1, int npart2)
2.  {
3.      for (int i = 1; i <= npart1 + npart2; ++i)
4.          used[i] = false;
5.      for (int i = 1; i <= npart1; ++i)
6.          if (matching[i] == undefined && !used[i])
7.              if (dfs(g, i))
8.                  return true;
9.      return false;
10. }
```

Логика функции *dfs()* заключается в поиске дополняющей цепи. Результатом работы функции является обновление массива *matching* вдоль найденной цепи, и как следствие, увеличение величины паросочетания на 1. Пример реализации функции *dfs* приведен в листинге 23.

Листинг 23. Функция dfs().

```
1.  bool dfs(tgraph g, int source)
2.  {
3.      used[source] = true;
4.      for (tedge *x = g[source]; x != 0; x = x->next)
6.          if (!used[x->destination])
7.          {
8.              used[x->destination] = true;
9.              int candidate = matching[x->destination];
10.             if (candidate == undefined || dfs(g, candidate))
10.             {
11.                 matching[source] = x->destination;
12.                 matching[x->destination] = source;
13.                 return true;
14.             }
25.         }
27.     return false;
28. }
```

В [12] показано, что алгоритмическая сложность данного алгоритма равняется $O(|V|*|E|)$, где $|V|$ – количество вершин в графе, $|E|$ – количество ребер в графе. Следует отметить, что существуют методики, которые позволяют снизить время выполнения данного алгоритма на некоторую константу. Одним из подходов является нахождение базового паросочетания путем применения жадных алгоритмов [1,4]. Несмотря на снижение времени выполнения алгоритма путем применения эвристик, алгоритм Куна подходит для нахождения максимального паросочетания для небольших графов.

4.3. Алгоритм Хопкрофта-Карпа

Алгоритм Хопкрофта-Карпа был впервые обоснован и представлен научной общественности Хопкрофтом и Карпом в 1973 году [13]. Как и в предыдущем разделе предположим, что нам задан двудольный граф $G = (V_1 \cup V_2, E)$. Алгоритм начинает свою работу, подобно алгоритму Диница, с построения функции расстояния для заданного графа. Как и в алгоритме Диница, данная функция может быть использована для проверки того, существует ли дополняющая цепь в графе относительно заданного паросочетания M. Построение же начинается из свободных вершин первой доли, листинг 24.

Листинг 24. Функция augment().

```
1.  bool augment(tgraph &g, int npart1, int npart2)
2.  {
3.      for (int i = 1; i <= npart1 + npart2; ++i)
4.          distance[i] = undefined;
5.      queue<int> q, transshipment;
6.      for (int i = 1; i <= npart1; ++i)
7.          if (matching[i] == undefined)
8.          {
9.              distance[i] = 0;
10.             q.push(i);
11.         }
12.     endings.clear();
13.     while (!q.empty())
14.     {
15.         for (; !q.empty(); q.pop())
16.         {
17.             int u1 = q.front();
18.             for (tedge *x = g[u1]; x != 0; x = x->next)
19.                 if (distance[x->destination] == undefined)
20.                 {
21.                     distance[x->destination] = distance[u1] + 1;
22.                     if (matching[x->destination] == undefined)
23.                         endings.push_back(x->destination);
24.                     else
25.                         transshipment.push(x->destination);
26.                 }
27.         }
```

```
28.         if (endings.size() > 0) return true;
29.         for (; !transshipment.empty(); transshipment.pop())
30.         {
31.             int u2 = transshipment.front();
32.             int u1 = matching[u2];
33.             distance[u1] = distance[u2] + 1;
34.             q.push(u1);
35.         }
36.     }
37.     return false;
38. }
```

Обратим внимание на цикл в строках 13-36. Внутри данного цикла происходит анализ вершин первой доли (строки 15-27) на предмет существования окончания дополняющей цепи, т.е. существования свободной вершины во второй доле. Если такие вершин существуют (строка 22), то они заносятся в массив *endings* (строка 23), иначе в массив *transshipment* (строка 25). Далее происходит анализ количества найденных свободных вершин. Если таковые существуют (строка 28), то процедура немедленно завершается с положительным результатом, и следующей итерацией алгоритма будет обновление текущего паросочетания с учетом найденных вершинно-непересекающихся дополняющих цепей [10,13]. В противном же случае в очередь добавляются вершины, которые являются парными для вершин из второй доли, записанных в массиве *transshipment* (строки 29-35).

Сами же пути, окончания которых записаны в массиве *endings,* можно построить, воспользовавшись процедурой поиска в глубину, листинг 25.

Листинг 25. Поиск дополняющей цепи.

```
1.  tpath dfs(tgraph &g, int ending)
2.  {
3.      if (!used[ending])
4.      {
5.          used[ending] = true;
6.          for (tedge *x = g[ending]; x != 0; x = x->next)
7.          if (distance[ending] == distance[x->destination] + 1 &&
8.              !used[x->destination])
9.          {
10.             if (distance[x->destination] == 0)
11.             {
12.                 used[x->destination] = true;
13.                 tpath result;
14.                 result.push_back(
15.                     make_pair(x->destination, ending));
16.                 return result;
17.             }
18.             tpath result = dfs (g, matching[x->destination]);
19.             if (result.size() > 0)
20.             {
```

```
21.                     result.insert(result.begin(),
22.                         make_pair(x->destination, ending));
23.                     return result;
24.                 }
25.             }
26.         }
27.     }
28.     tpath result;
29.     return result;
30. }
```

Отметим, что процедура *dfs()* анализирует только допустимые дуги, т.е. такие дуги (u,v), для которых $d(v)=d(u)+1$ (строка 7). Следует сказать, что вершина *ending* в параметрах процедуры *dfs()* всегда принадлежит одной и той же доле (пусть, для определенности, второй). Это поясняется рекурсивным вызовом в строке 18. Перед непосредственным выполнением строки 18, происходит поиск вершины *destination*, непосредственно связанной с *ending*. Так как граф двудольный, то вершина *destination* будет принадлежать первой доле. Если расстояние до вершины равняется 0, то поиск дополняющей цепи закончен, и процедура немедленно возвращается (строка 16). В противном случае, осуществляется переход из вершины *destination* во вторую долю по ребру $(destination,matching[destination])$. Далее происходит поиск оставшейся части цепи, но уже из вершины *matching*[*destination*] (строка 18). Так как процедура *dfs()* просматривает каждую вершину один раз (строка 3), то и пути, возвращаемые процедурой, не будут содержать повторяющихся вершин [10,13]. Объединив обе процедуры вместе, получим реализацию алгоритма Хопкрофта-Капра, приведенную в листинге 26.

Листинг 26. Алгоритм Хопкрофта-Карпа.

```
1.  int hopcroft_karp_algorithm(tgraph &g, int npart1, int npart2)
2.  {
3.      int cardinality = 0;
4.      for (int i = 1; i <= npart1 + npart2; ++i)
5.          matching[i] = undefined;
6.      while (augment(g, npart1, npart2))
7.      {
8.          for (int i = 1; i <= npart1 + npart2; ++i)
9.              used[i] = false;
10.         for (size_t i = 0; i < endings.size(); ++i)
11.             improve_matching (matching, dfs(g, endings[i]));
12.     }
13.     for (int i = 1; i <= npart1; ++i)
14.         if (matching[i] != undefined)
15.             cardinality = cardinality + 1;
16.     return cardinality;
17. }
```

Ключевой здесь является строка 11, в которой происходит увеличение паросочетания вдоль найденной допустимой цепи. Как уже говорилось, такая цепь строится функцией *dfs()*. Логика улучшения найденного паросочетания выглядит следующим образом, листинг 27.

Листинг 27. Процедура improve_matching().

```
1.  void improve_matching(tmatching &matching, tpath &path)
2.  {
3.      for (size_t j = 0; j < path.size(); ++j)
4.      {
5.          int u1 = path[j].first;
6.          int u2 = path[j].second;
7.          matching[u1] = u2;
8.          matching[u2] = u1;
9.      }
10. }
```

В [10,13] показано, что алгоритмическая сложность данного алгоритма равняется $O(\sqrt{|V|}*|E|)$, где $|V|$ – количество вершин в графе, $|E|$ – количество ребер в графе. На текущий день алгоритм Хопкрофта-Карпа является асимптотически лучшим алгоритмом нахождения наибольшего паросочетания в двудольных графах. Помимо теоретической значимости, данный алгоритм показывает высокие результаты и на практике. Его применение оправдано на графах с количеством вершин и ребер до $50,000$ включительно.

4.4. Заключение

В заключение данного раздела дадим краткий обзор задач теории графов , которые эффективно решаются путем применения алгоритмов нахождения максимального паросочетания.

Определение 36. *Деревом* называется связный граф $T=(V,E)$, в котором между любой парой вершин u и v существует ровно один путь.

Любое дерево можно преобразовать в двудольный граф. Для этого достаточно запустить поиск в ширину, либо поиск в глубину, для одной из вершин дерева. В процессе обхода вершины будут последовательно помечаться либо 0, либо 1, где 0 означается принадлежность вершины первой доле, а 1 – второй. Пример дерева и соответствующего двудольного графа приведен на рис. 27.

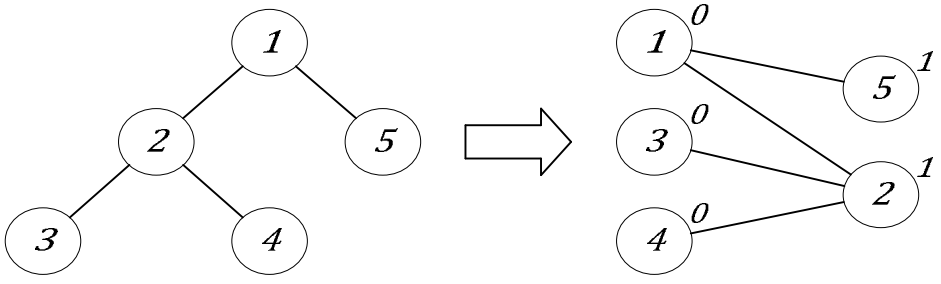


Рис. 27. Пример дерева и полученного двудольного графа.

Как следствие, задача построения максимального паросочетания на дереве может быть решена одним из алгоритмов, рассмотренных ранее.

Определение 37. *Вершинным покрытием* неориентированного графа $G=(V,E)$ называется подмножество $V'\subset V$ такое, что для любого ребра $(u,v)\in E$ хотя бы одна из вершин принадлежит V'.

Определение 38. Вершинное покрытие V' неориентированного графа $G=(V,E)$ называется минимальным, если не существует такого вершинного покрытия V'' такого, что $|V''|<|V'|$.

Задача поиска минимального вершинного покрытия заданного неориентированного графа $G=(V,E)$ в общем случае является NP-полной [4]. В то же самое время существуют полиномиальные алгоритмы поиска минимального вершинного покрытия для двудольных графов. Такие алгоритмы в большинстве случаев опираются на теорему 4, которая была сформулирована и доказана Кенигом в 1916 году [1].

Теорема 4. В двудольном графе $G=(V_1\cup V_2,E)$ величина максимального паросочетания равняется числу вершин в минимальном вершинном покрытии.

В [1,2] приводится один аз алгоритмов преобразования максимального паросочетания в минимальное вершинное покрытие.

Как уже отмечалось, алгоритм Хопкрофта-Карпа находит свое применение лишь в задачах большой размерности. Его же применение на практике обусловлено сложностью кодирования и последующей отладкой. В большинстве же случаев алгоритма Куна вполне достаточно для решения задач, прямо или косвенно связанных с нахождением максимальных паросочетаний

4.5. Примеры решения задач

Задача 1. Всем известно, что улицы города Нью-Йорка расположены перпендикулярно друг к другу. Поэтому расстояние между двумя перекрестками в этом городе вычислить довольно просто. Если вы находитесь на перекрестке с координатами (x_1,y_1), то расстояние, которое требуется пройти до перекрестка (x_2,y_2) будет равняться $c*(|x_1-x_2|+|y_1-y_2|)$, где c – некоторый известный коэффициент.

Предположим, что в городе Нью-Йорке работает N курьеров, каждый из которых может передвигаться со скоростью v_i, $i=\overline{1,N}$. В начальный момент времени t_0 i-ый курьер находится на перекрестке с координатами (x_i^d,y_i^d)[6]. К тому же вам известно, что каждый из курьеров имеет при себе один артефакт, который для вас представляет ценность.

Вы решили собрать артефакты, которыми владеют курьеры. Для этого в город приехало M специалистов, которые в момент времени t_0 находятся в

[6] Чтобы не вводить читателя в заблуждение, поясним, что верхний индекс *d* не несет в себе никакой смысловой нагрузки, а происходит всего лишь от английского *delivery service*, что в переводе означает *служба доставки.*

точках с координатами (x_i^s, y_i^s).Каким максимальным количеством артефактов вы сможете завладеть, если у курьеров есть всего лишь Δt единиц времени на доставку материалов вашим людям.

Решение. Данная задача является одним из типичных «представителей» круга задач, которые можно эффективно решать рассмотренными выше алгоритмами, т.е. алгоритмами нахождения максимального паросочетания в графе. Чтобы это показать, построим следующий граф $G = (V_d \cup V_s, E)$, в котором множество V_d обозначает сотрудников службы доставки, а множество V_s – специалистов. Соединим ребром курьера $i^d (i = \overline{1, N})$ со специалистом $j^s (j = \overline{1, M})$, если курьер i^d успевает доставить артефакт специалисту j^s за отведенное время Δt, т.е. если $c * (|x_i^d - x_j^s| + |y_i^d - y_j^s|) \le v_i * \Delta t$. Величина максимального паросочетания в построенном графе и будет значением наибольшего количества артефактов, которым вы сможете завладеть.

Следует отметить, что если бы каждый из курьеров владел большим количеством артефактов, то алгоритм решения данной задачи отличался бы от алгоритма, рассмотренного выше.

4.6. Задачи для самостоятельного решения

Задача 3. Рассмотрим стоянку для автомобилей, представляющую собой прямоугольную площадку размерами $n \times m$, разделенную на $n * m$ мест. В некоторый момент времени часть мест является недоступной, часть же мест является свободной. Вам задан план площадки, на котором вы можете определить позиции своих автомобилей и позиции парковочных мест, которые вам предоставлены на сегодня. Пример площадки приведен на рис.28. Также вам известно, что передвижение автомобиля из одного места на другое требует Δt единиц времени, причем передвигаться разрешается только в свободное соседнее место по вертикали, либо горизонтали. Вы хотели бы узнать минимальное количество времени, которое потребуется для того, чтобы расставить все автомобили на парковочные места, либо убедиться в том, что этого сделать нельзя.

#	1	2	3	4	5	6	7	8
1	X	X	.	**П**	X	X	**П**	.
2	X	**М**	.	X	.	.	X	.
3	X	.	.	.	**М**	.	.	.
4	X	**М**	.	X	X	.	X	**П**

Рис. 28. Пример площадки.
“**X**” – занятая позиция, “**.**” – свободная позиция,
“**М**” – позиция автомобиля, “**П**” – парковочное место.

Для приведенного примера минимальное затраченное время будет равняться 8 единицам. Отметим, что количество парковочных мест может отличаться от количества автомобилей.

Задача 4. Рассмотрим комнату, план которой представляет собой прямоугольник размерами $n \times m$, состоящий из $n * m$ секторов. Некоторые из секторов комнаты являются *стенами*, в некоторых же секторах находятся *ловушки*. Оставшиеся сектора являются свободными. Строки и столбцы данного плана будем называть *коридорами*.

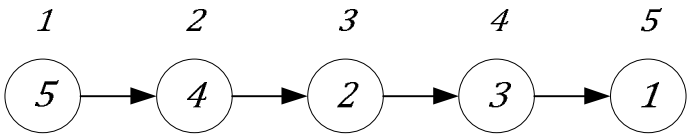
Предположим, что два человека находятся в позициях (i_1, j_1) и (i_2, j_2) соответственно, причем данные позиции отличны от стен и ловушек. Тогда эти два человека могут видеть друг друга, если 1) они находятся в одном коридоре и 2) между ними нет стен.

Перед вами поставлена задача охраны комнаты. Для этих целей вы можете расставить в некоторых позициях комнаты охранников таким образом, чтобы 1) все коридоры были защищены и 2) никакие два охранника не могли видеть друг друга. Определите минимальное количество охранников, достаточное для того, чтобы комната оставалась в безопасности.

Задача 5. Рассмотрим однопроцессорную машину M, которая умеет выполнять примитивные задачи. Вам задан пакет J, состоящий из N примитивных задач. Так как машина M является однопроцессорной, то задачи она выполняет строго последовательно. Для каждой задачи j пакета J известен номер $r = r[j,k]$, $1 \le r \le N$, $k = \overline{1,N}$ такой, что для выполнения задачи j r-ой по счету требуется k ячеек памяти. Вам требуется определить минимальное количество ячеек памяти, которое должна зарезервировать машина для успешного выполнения задач из пакета J. Для примера рис. 29. ответом будет 3.

r	*1*	*2*	*3*	*4*	*5*
1	4	5	2	3	1
2	3	4	5	1	2
3	2	5	4	3	1
4	4	3	2	5	1
5	2	3	1	5	4

а)



б)

Рис. 29. а) Пакет J состоящий из 5 задач. б) оптимальная последовательность выполнения задач, с количеством ячеек памяти равным 3

Литература

1. Ahuja R.K. Network Flows: Theory, Algorithms, and Applications / R.K. Ahuja, T.L. Magnanti, J.B. Orlin – USA: Prentice Hall, 1993 – 864p.

2. Пападимитриу Х. Комбинаторная оптимизация. Алгоритмы и сложность. / Х. Пападимитриу, К. Стайглиц, пер. с англ. – М.: Мир, 1984 – 512с.

3. Скиена С. С. Олимпиадные задачи по программированию. Руководство по подготовке к соревнованиям. / С.С. Скиена, М.А. Ревилла, пер. с англ. – М.: Кудиц-Образ, 2005 – 416с.

4. Кормен Т. Алгоритмы. Построение и анализ. / Т. Кормен, Ч. Лейзерсон, Р. Ривест, К. Штайн, пер. с англ. – М.: Вильямс, 2007 г. – 1296с.

5. Goldberg A.V. Network flow algorithms. / A.V. Goldberg, E. Tardos, R.E. Tarjan – Springer-Verlag, Algorithms and Combinatorics 9, 1990 – 101-164pp.

6. Ахо А., Структуры данных и алгоритмы. / А. Ахо, Дж. Ульман, Дж. Хопкрофт, пер. с англ. – М: Вильямс, 2003 г – 384с.

7. Ford L.R. Maximal flow through a network / L.R. Ford, D.R. Fulkerson – Canadian Journal of Mathematics 8, 1956 – 399-404pp.

8. Edmonds J. Theoretical improvements in algorithmic efficiency for network flow problems / J. Edmonds, R.M. Karp – Journal of ACM 19, 1972 – 248-264pp.

9. Dinic E.A. Algorithm for solution of a problem of maximum flow in a network with power estimation. / E.A. Dinic – Soviet Math Doklady 11, 1970 – 1277-1280pp.

10. Липский В. Комбинаторика для программистов. / В. Липский, пер. с пол. – М.: Мир, 1988 – 213с.

11. Berge C. Two theorems in graph theory. / C. Berge – Proceedings of the National Academy of Sciences USA 43, 1957 –. 842-844pp.

12. Kuhn H.W. The Hungarian method for the assignment problem. / H.W. Kuhn, Naval Research Logistics Quarterly 2, 1955 – 83-97pp.

13. Hopcroft J.E. A $n^{2.5}$ algorithm for maximum matchings in bipartite graphs. / J.E. Hopcroft, R.M. Karp – SIAM Journal on Computing 2, 1973 – 225-231pp.

Св. план 2010, поз.54

Учебное издание

Актанорович Сергей Владимирович
Волосевич Алексей Александрович
Сиротко Сергей Иванович

АЛГОРИТМЫ И СТРУКТУРЫ ДАННЫХ. ПОТОКОВЫЕ АЛГОРИТМЫ

Методическое пособие
по курсу «Теория графов. Потоковые алгоритмы»
для студентов специальности I-31 03 04 «Информатика»
всех форм обучения

Редактор
Корректор

Подписано в печать
Гарнитура «Таймс».
Уч.-изд. л. .

Формат 60x84 1/16.
Печать ризографическая
Тираж 100 экз.

Бумага офсетная.
Усл. печ. л.
Заказ 548.

Издатель и полиграфическое исполнение: Учреждение образования
«Белорусский государственный университет информатики и радиоэлектроники».
ЛИ № 02330/0056964 от 01.04.2004.
ЛП № 0233/0131666 от 30.04.2004.
220013, Минск, П. Бровки, 6