Медведев Михаил Геннадиевич
Кандидат физико-математических наук, доцент факультета кибернетики Киевского национального университета имени Тараса Шевченко.

Алгоритм Дейкстры и его реализация средствами STL

Описывается алгоритм решения задачи поиска кратчайшего пути из одного источника до остальных вершин графа, именуемый алгоритмом Дейкстры. Рассматривается реализация алгоритма с помощью массивов, STL-контейнеров – очереди с приоритетами priority_queue, множества set, а также с использованием операций над кучей push_heap и pop_heap.

Задача. Пусть есть страна G, в которой есть множество городов (обозначим это множество как V), и множество дорог, соединяющих пары городов (обозначим их как E). Не факт, что каждая пара городов соединена дорогой. Иногда, чтобы добраться из одного города в другой, следует посетить несколько транзитных городов. У дорог есть длина. В стране G есть город-столица s. Необходимо найти кратчайшие пути из столицы до всех остальных городов.

Приведём *математическую формулировку* этой задачи.

Пусть $G=(V, E)$ – ориентированный граф, каждое ребро которого помечено неотрицательным числом (*вес ребра*). Пометим некоторую вершину s и назовём её *истоком*. Найти кратчайшие пути от истока s до всех других вершин графа G.

Эта задача имеет название *«Поиск кратчайших путей с одним истоком»*.

Любая часть кратчайшего пути сама есть кратчайший путь. Это позволяет для решения задачи применить *динамическое программирование*.

Метод динамического программирования применяется к задачам, которые лежат в каком-либо множестве однотипных задач, и некоторые из задач этого множества простые и имеют очевидные решения, и кроме того, есть возможность находить решения новых задач, основываясь на имеющихся решениях. Так, в нашем случае задача «найти кратчайший путь из A в B»

является одной из задач вида «найти кратчайший путь из A в X». Одна из этих задач имеет очевидное решение «найти кратчайший путь из A в A» – никуда идти не нужно, длина кратчайшего пути равна 0. Осталось изобрести способ пошагово расширять множество решённых задач указанного вида.

Про задачи, решаемые методом динамического программирования, говорят, что они имеют *оптимальную подструктуру* и распадаются на множество перекрывающихся подзадач. Оптимальность подструктуры означает, что оптимальное решение подзадач меньшего размера может быть использовано для решения всей задачи. Как правило, при решении таких задач изначально выбирается множество параметров, по которым ведётся вычисление. Связь между параметрами и подзадачами устанавливается при помощи рекуррентных соотношений. В процессе обработки подзадач используется *запоминание их промежуточных результатов* (обычно при помощи массивов).

Наша задача о кратчайших путях обладает необходимым свойством оптимальности для подзадач.

Лемма. Пусть $G=(V, E)$ – взвешенный ориентированный граф с весовой функцией $w: E \to R^+$. Если $p=(v_1, v_2, \dots, v_k)$ – кратчайший путь из v_1 в v_k и $1 \le i \le j \le k$, то $p_{ij} = (v_i, v_{i+1}, \dots, v_j)$ есть кратчайший путь из v_i в v_j.

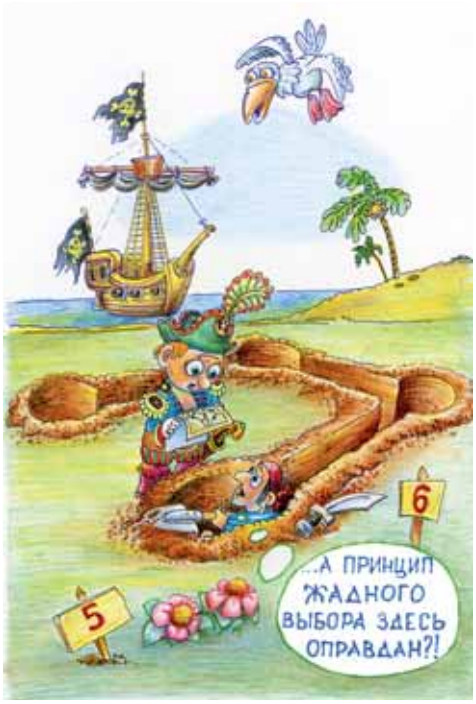
Обозначим через $\delta(u, v)$ величину кратчайшего пути между вершинами u и v. Вес ребра между вершинами x и y будем обозначать $w(x, y)$.

Следствие. Рассмотрим кратчайший путь p из s в v. Пусть $u \to v$ – последнее ребро этого пути. Тогда $\delta(s, v)=\delta(s, u)+w(u, v)$.

Лемма. Пусть $G=(V, E)$ – взвешенный ориентированный граф с весовой функцией $w: E \to R^+$. Пусть $s \in V$. Тогда для всякого ребра $(u, v) \in E$, имеет место неравенство:

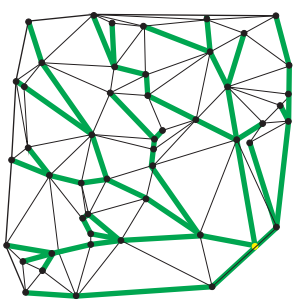
$$\delta(s, v) \le \delta(s, u)+w(u, v).$$

Для решения задачи кратчайшего пути с одним истоком воспользуемся *жадным алгоритмом Дейкстры*. Принцип жадного выбора здесь оправдан, так как последовательность локально оптимальных выборов даёт глобально оптимальное решение.



Алгоритм Дейкстры строит множество S вершин, для которых кратчайшие пути от истока уже известны. На каждом шаге к множеству S добавляется та из вершин v, расстояние до которой от истока не

больше, нежели до остальных вершин. Такое добавление вершины v как раз и характеризует принцип жадного выбора. После добавления v в S кратчайшее расстояние от истока до v уже никогда не улучшится, установим used[v] = 1. Поскольку веса дуг неотрицательны, то кратчайший путь от истока до конкретной вершины из множества S будет проходить только через вершины из S. Такой путь будем называть *особенным*. На каждом шаге алгоритма используется массив d, в который записываются длины кратчайших особенных путей для каждой вершины. Когда множество S будет содержать все вершины графа (для всех вершин будут найдены особенные пути), тогда массив d будет содержать длины кратчайших путей от истока до каждой вершины.



Вход. Первая строка содержит количество вершин n в графе и номер вершины – истока s. Каждая

следующая строка описывает ориентированное ребро графа и содержит номера вершин, которые оно соединяет, а также его вес. Вершины графа нумеруются числами от 1 до n.

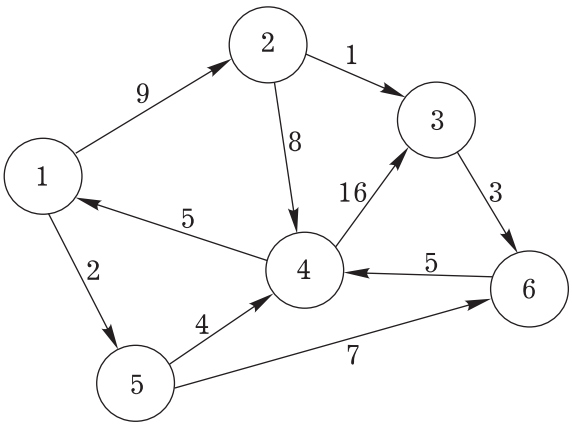
Выход. Для каждой вершины v графа вывести строку

```
From source s to destination v distance is dist
```

Здесь *dist* – кратчайшее расстояние от истока s до вершины v. Строки выводить в порядке возрастания номеров вершин v.

Пример входа	Пример выхода
6 4	From source 4 to destination 1 distance is 5
1 2 9	From source 4 to destination 2 distance is 14
1 5 2	From source 4 to destination 3 distance is 15
2 3 1	From source 4 to destination 4 distance is 0
2 4 8	From source 4 to destination 5 distance is 7
3 6 3	From source 4 to destination 6 distance is 14
4 1 5	
4 3 16	
5 4 4	
5 6 7	
6 4 5	

Граф, приведённый в тестовом примере, имеет вид:



Реализация алгоритма Дейкстры при помощи массивов

Считаем, что вершины графа G помечены целыми числами, среди которых выделен исток. Элемент матрицы $m[i][j]$ содержит вес ребра (i, j). Если ребра (i, j) не существует, то $m[i][j]$ положим равным максимальному числу. Положим used$[i]$ = 1, если вершина i уже вошла во множество S. Вершина с номером *source* является истоком.

```
#include <stdio.h>
#include <memory.h>
#define MAX 100
#define INF 2100000000

int i,j,n,source;
int b,e,dist,v;
int m[MAX][MAX],used[MAX],d[MAX];

// Инициализация массивов, используемых в алгоритме.
void Init(void)
{
  memset(m,63,sizeof(m));
  memset(used,0,sizeof(used));
  memset(d,63,sizeof(d)); d[source] = 0;
}

// Релаксация ребра (i, j).
void Relax(int i, int j)
{
  if (d[j] > d[i] + m[i][j]) d[j] = d[i] + m[i][j];
}

// Поиск номера ещё неиспользованной вершины с
// наименьшим расстоянием от истока (значением d[i]).
int Find_Min(void)
{
```

```
  int i,v,min = INF / 2;
  for(i=1;i<=n;i++)
    if (!used[i] && d[i] < min) min = d[i], v = i;
  return v;
}

int main(void)
{
```

Инициализируем массивы, читаем данные входного графа. Изначально расстояния от вершины *source* до любой другой положим равными плюс бесконечности. При этом d[*source*] = 0, так как расстояние от *source* до *source* равно нулю. Изначально множество S состоит из одной вершины *source*.

```
scanf("%d %d",&n,&source);
Init();
while(scanf("%d %d %d",&b,&e,&dist) == 3) m[b][e] = dist;
```

Совершаем $n-1$ цикл. На каждой итерации к множеству S добавляется вершина v, расстояние до которой от истока не больше, нежели до остальных ещё не включенных в S вершин. Проводим релаксацию всех рёбер, выходящих из вершины v. Длина кратчайшего расстояния от истока до v уже никогда не улучшится, добавляем v к S, устанавливаем used[v] = 1.

```
for(i = 1; i < n; i++)
{
  v = Find_Min();
  for(j = 1; j <= n; j++)
    if (!used[j]) Relax(v,j);
  used[v] = 1;
}
```

Выводим результат. Ячейка d[i] содержит кратчайшее расстояние от вершины *source* до i.

```
 for(i = 1; i <= n; i++)
     printf("From source %d to destination %d distance is %d\n",
                  source, i, d[i]);
 return 0;
}
```

Пример. Рассмотрим граф, приведённый в тестовом примере. Вершина 4 является истоком. Инициализируем $S = \{\}$. Для каждого значения k положим $d[k]$ равным максимальному натуральному числу. При этом $d[4] = 0$, поскольку расстояние от вершины до её самой равно 0.

На первой итерации находим наименьшее значение среди $d[i]$, где i – вершина, ещё не вошедшая в S. $\min\{ d[1], d[2], d[3], d[4], d[5], d[6] \} = d[4] = 0$. Первой во множество S

будет включена вершина 4. Пересчитываем значения $d[i]$ для всех $i \in V \setminus S = \{1, 2, 3, 5, 6\}$:

$$d[1] = \min(d[1], d[4] + m[4][1]) = \\ = \min(+\infty, 0 + 5) = 5;$$

$$d[2] = \min(d[2], d[4] + m[4][2]) = \\ = \min(+\infty, 0 + \infty) = +\infty;$$

$$d[3] = \min(d[3], d[4] + m[4][3]) = \\ = \min(+\infty, 0 + 16) = 16;$$

$$d[5] = \min(d[5], d[4] + m[4][5]) = \\ = \min(+\infty, 0 + \infty) = +\infty;$$

$$d[6] = \min(d[6], d[4] + m[4][6]) = \\ = \min(+\infty, 0 + \infty) = +\infty.$$

Рассмотрим вторую итерацию. Ищем минимум среди $d[i]$, где $i \notin S$: $\min\{ d[1], d[2], d[3], d[5], d[6] \} = d[1] = 5$. Второй во множество S будет включена вершина 1, то есть $S = \{1, 4\}$. Пересчитываем значения $d[i]$ для всех $i \in V \setminus S = \{2, 3, 5, 6\}$:

$$d[2] = \min(d[2], d[1] + m[1][2]) = \\ = \min(+\infty, 5 + 9) = 14;$$

$$d[3] = \min(d[3], d[1] + m[1][3]) = \\ = \min(16, 5 + \infty) = 16;$$

$$d[5] = \min(d[5], d[1] + m[1][5]) = \\ = \min(+\infty, 5 + 2) = 7;$$

$$d[6] = \min(d[6], d[1] + m[1][6]) = \\ = \min(+\infty, 5 + \infty) = +\infty.$$

Результат выполнения всех итераций приведён в таблице. Вершина v, которая выбирается и добавляется к S на каждом шаге, выделена и подчёркнута. Значения $d[i]$, для которых $i \in S$, выделены курсивом.

итерация	S	$d[1]$	$d[2]$	$d[3]$	$d[4]$	$d[5]$	$d[6]$
начало	{}	$+\infty$	$+\infty$	$+\infty$	**0**	$+\infty$	$+\infty$
1	{4}	**5**	$+\infty$	16	*0*	$+\infty$	$+\infty$
2	{1, 4}	*5*	14	16	*0*	**7**	$+\infty$
3	{1, 4, 5}	*5*	14	16	*0*	*7*	**14**
4	{1, 4, 5, 6}	*5*	**14**	16	*0*	*7*	*14*
5	{1, 2, 4, 5, 6}	*5*	*14*	**15**	*0*	*7*	*14*

Итеративный процесс алгоритма Дейкстры

Далее рассмотрим реализацию алгоритма Дейкстры при помощи различных контейнеров и функций, описанных в библиотеке шаблонов STL.

STL (Standart Template Library) – это набор шаблонов функций и классов в языке C++, включающий в себя различные контейнеры данных (список, очередь, множество, отображение, хэш таблица, очередь с приоритетами) и базовые алгоритмы (сортировка, поиск). Историю появления и основные понятия библиотеки шаблонов можно найти, например, в Википедии:

http://en.wikipedia.org/wiki/Standard_Template_Library. Детальное описание STL можно найти на странице http://www.sgi.com/tech/stl.

Реализация алгоритма Дейкстры при помощи очереди с приоритетами

Реализуем алгоритм Дейкстры при помощи очереди с приоритетами. Эта структура данных поддерживается библиотекой шаблонов STL и называется priority_queue. Она позволяет хранить пары (ключ, значение) и достаточно быстро выполнять две операции:

- вставка элемента с назначенным приоритетом;
- извлечение элемента с наивысшим приоритетом.

Объявим очередь с приоритетами *pq*, элементами которой будут пары (*distance*, *node*), где *distance* – расстояние от истока до вершины *node*. При вставке элементов в голове очереди всегда будет находится пара (*distance*, *node*) с наименьшим значением *distance*. Таким образом, номер неиспользованной вершины, расстояние до которой от истока минимально, доступен как pq.top().second.

```
#include <cstdio>
#include <queue>
#define MAX 10000
using namespace std;

priority_queue<pair<int,int>, vector<pair<int,int> >,
               greater<pair<int,int> > > pq; //(distance,node)
vector<vector<int> > m(MAX, vector<int> (MAX));
vector<int> dist(MAX, 0x3FFFFFFF);
int n,src;

int main(void)
{
  int i,a,b,d;
  scanf("%d %d", &n, &src);
  while(scanf("%d %d %d", &a, &b, &d) == 3) m[a][b] = d;

  pq.push(make_pair(0,src)); dist[src] = 0;
  while(!pq.empty())
  {


    pair<int,int> s = pq.top();pq.pop();
    for(i = 1; i <= n; i++)
      if (m[s.second][i] && (dist[i] > dist[s.second] +
      m[s.second][i]))
        dist[i] = dist[s.second] + m[s.second][i],
        pq.push(make_pair(dist[i],i));
  }
  for(i = 1; i <= n; i++)
    printf("From source %d to destination %d distance is %d\n",
```

```
            src, i, dist[i]);
  return 0;
}
```

Реализация алгоритма Дейкстры при помощи множества

Промоделируем очередь с приоритетами при помощи множества set<pair<int,int> > *s*. Его элементами являются пары, второй элемент которых содержит номер вершины *i*, а первый – текущее оптимальное расстояние от истока до вершины *i* (значение *d*[*i*]).

Преимущество такого хранения данных состоит в том, что элементы множества *s* всегда отсортированы по первой компоненте пары. То есть пара с вершиной, добавляемой каждый раз во множество *S*, находится наверху *s* и доступна как *s.begin().

```
#include <cstdio>
#include <set>
#include <memory>
#define MAX 10000
#define INF 0x3F3F3F3F
using namespace std;

typedef pair<int, int> ii;

int i, v, n, dist, source, b, e;
int m[MAX][MAX], d[MAX];
set<ii> s;

int main(void)
{
  scanf("%d %d", &n, &source);
  memset(m, 63, sizeof(m));
  while(scanf("%d %d %d", &b, &e, &dist) == 3) m[b][e] = dist;
  memset(d, 0x3F, sizeof(d)); d[source] = 0;
  s.insert(ii(0,source));
```

Тело цикла while выполняется *n* раз (*n* – количество вершин в графе). На каждой итерации одна и только одна вершина заносится во множество *S* и убирается из дальнейшего рассмотрения. На последней, *n*-ой итерации, никакое ребро не релаксирует, а из множества *s* убирается информация о последней вершине.

```
  while(!s.empty())
  {
    ii top = *s.begin(); s.erase(s.begin());
    v = top.second;
```

Вершина *v* добавляется на текущем шаге во множество *S*. Производим релаксацию рёбер, ведущих из *v*. Если ребро (v, i) релаксирует, то следует убрать из *s* пару $(d[i], i)$ и добавить $(d[v] + m[v][i], i)$.

```
    for(i = 1; i <= n; i++)
      if (d[i] > d[v] + m[v][i])
      {
        if (d[i] != INF) s.erase(s.find(ii(d[i],i)));
        d[i] = d[v] + m[v][i];
        s.insert(ii(d[i],i));
      }
  }

  for(i = 1; i <= n; i++)
    printf("From source %d to destination %d distance is %d\n",
           source, i, d[i]);
  return 0;
}
```

Реализация алгоритма Дейкстры при помощи кучи

Очередь с приоритетами, как известно, можно реализовать при помощи кучи. Вместо непосредственного объявления очереди с приоритетами будем использовать вектор пар *pq*, а при операциях вставки/извлечения его элементов будем пользоваться функциями библиотеки шаблонов для поддержания основного свойства кучи pop_heap и push_heap. Основное свойство кучи поддерживается на множестве элементов от *pq*[0] до *pq*[*len*].


```
#include <cstdio>
#include <queue>
#include <algorithm>
#define MAX 10000
using namespace std;

vector<pair<int,int> > pq(MAX); //(distance,node)
vector<vector<int> > m(MAX, vector<int> (MAX));
vector<int> dist(MAX,0x3FFFFFFF);
int n, src;

int main(void)
{
  int i, a, b, d, len;
  scanf("%d %d",&n,&src);
  while(scanf("%d %d %d", &a, &b, &d) == 3) m[a][b] = d;

  pq[0] = make_pair(0,src); len = 1; dist[src] = 0;
  while(len)
```

```
  {
    pair<int,int> s = pq[0];
    pop_heap(pq.begin(),pq.begin()+len, greater<pair<int,int>
    >()); len--;

    for(i = 1; i <= n; i++)
      if (m[s.second][i] && (dist[i] > dist[s.second] +
      m[s.second][i]))
      {
        dist[i] = dist[s.second] + m[s.second][i];
        pq[len] = make_pair(dist[i],i); len++;
        push_heap(pq.begin(),pq.begin()+len,greater<pair<int,
        int> >());
      }
  }
  for(i = 1; i <= n; i++)
    printf("From source %d to destination %d distance is %d\n",
           src, i, dist[i]);
  return 0;
}
```

Время работы алгоритма Дейкстры

При использовании массивов алгоритм поиска кратчайших путей требует выполнения $n - 1$ итерации, в каждой из которой поиск вершины v и релаксация рёбер выполняется за время $O(n)$. Таким образом, полное время выполнения алгоритма составляет $O(n^2)$.

Для поиска кратчайших путей на *разреженных графах* целесообразно воспользоваться очередью с приоритетами. Время выполнения алгоритма равно $O((|V| + |E|) \lg|V|) = O(|E| \lg|V|)$, где $|V|=n$ есть число вершин, а $|E|$ – число рёбер в графе.

В олимпиадных соревнованиях задача поиска путей достаточно распространена. Например, при решении следующих задач требуется реализовать алгоритм Дейкстры:

[Тянь-Шань] http://acm.tju.edu.cn/toj: 2134 (106 miles to Chicago), 2819 (Travel), 2870 (The K-th City).

[Топкодер] www.topcoder.com: SRM 241 (AirTravel).

Список литературы

1. «Алгоритмы: построение и анализ», Кормен Т., Лейзерсон Ч., Ривест Р., Штайн К. – Москва, Санкт-Петербург, Киев, 2005, 1292 с.

2. «Практика и теория программирования», Книга 2. Винокуров Н.А., Ворожцов А.В. – М: Физматкнига, 2008, 288 с.

3. Статья «Бинарная пирамида», журнал «Потенциал» №7, 2007.