Основы программирования

Пути на графах

Поиск кратчайших путей в графе

Пусть $G = (V, E)$ – взвешенный ориентированный граф с матрицей весов дуг $c_{ij} \geq 0,\ i, j = 0 \ldots n-1,\ c_{ij} = \infty,$ если $(i, j) \notin E$.

Если в графе существует некоторый путь, проходящий через вершины $s, i, j, \ldots, k, t,$ то его весом является величина $c_{si} + c_{ij} + \cdots + c_{kt}$. Если существует несколько различных путей из s в t, то можно сформулировать задачу поиска кратчайшего (минимального по весу) пути.

3 варианта задачи поиска кратчайших путей в графе:

- кратчайший путь между парой вершин s и t,
- кратчайшие пути из одной вершины s во все остальные,
- кратчайшие пути между всеми парами вершин.

Алгоритм Дейкстры

Алгоритм Дейкстры позволяет найти кратчайшие пути из одной вершины-источника во все остальные вершины взвешенного ориентированного графа.

Вес кратчайшего пути из s в t будем обозначать d_{st}.

Пусть кратчайший путь проходит через вершины $s, i, \ldots, u, t$, тогда $d_{st} = d_{su} + c_{ut}$ (это можно легко доказать от противного).

Следовательно, для задачи поиска кратчайшего пути выполняется свойство оптимальности подзадач.

Алгоритм Дейкстры

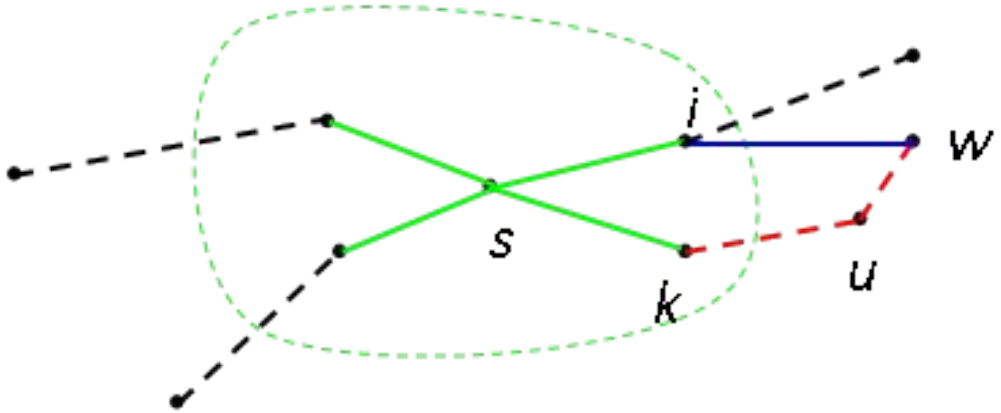
Пусть для некоторого мн-ва вершин V_0 (**старых**) найдены кратчайшие пути из s с весами $d_{si},\ i \in V_0$ (V_0 никогда не пусто, т.к. $s \in V_0,\ d_{ss} = 0$).

Для остальных вершин $j \notin V_0$ (**новых**) проверка пока не завершена, но определены текущие значения длин путей: $\hat{d}_{sj} = \min(d_{si} + c_{ij}):\ i \in V_0$, т.е. соответствующий путь проходит только через старые вершины.

При $V_0 = \{s\}$ начальные значения $\hat{d}_{sj} = c_{ij}$.

Если в графе еще не найдены пути из s в j, проходящие только через старые вершины, то $\hat{d}_{sj} = \infty$.

Алгоритм Дейкстры



Выберем $\hat{d}_{sw} = \min(\hat{d}_{sj})$ и соответствующую вершину w. По условию выбора, любой путь, проходящий из s в w через произвольную новую вершину u, не может иметь вес, меньший чем $\hat{d}_{sw}$ (вес такого пути $\geq \hat{d}_{su} \geq \hat{d}_{sw}$).

Поэтому $\hat{d}_{sw} = d_{sw}$ (т.е. текущий путь является кратчайшим), и вершину w можно включать во множество старых вершин. После этого необходимо скорректировать текущие длины путей для оставшихся новых вершин j:

$\hat{d}_{sj} = \min(\hat{d}_{sj},\ d_{sw} + c_{wj})$.

Алгоритм Дейкстры

Процесс поиска кратчайших путей продолжается, пока все вершины графа не станут старыми. Поиск можно прекратить, если на некотором шаге все $\hat{d}_{sj} = \infty$ – это означает, что оставшиеся новые вершины недостижимы из источника s.

$\hat{d}_{sj} = \min(d_{si} + c_{ij}) : \ i \in V_0$, т.е. для каждой новой вершины на каждом шаге определена старая вершина – предыдущая в пути. Сохраняя ссылки на предыдущие вершины, можно получать не только веса путей d_{sj}, но и соответствующие им последовательности пройденных вершин графа.

Алгоритм Дейкстры

Для добавления к V_0 вершины w производится локально оптимальный выбор $\hat{d}_{sw} = \min(\hat{d}_{sj})$: $j \notin V_0$, не закрывающий путь к оптимальному решению для остальных новых вершин.

Таким образом, выполнены условия, позволяющие реализовать эффективный жадный алгоритм поиска всех кратчайших путей из одной вершины-источника.

Приведенный ниже метод **`minpath`** реализует алгоритм Дейкстры. Он получает в качестве параметров и формирует массивы, которые должны быть выделены заранее:

`double D[vernum]` – текущие/кратчайшие веса путей из источника **`s`**,

`int P[vernum]` – номера предыдущих вершин в пути.

Методы WGraph для алгоритма Дейкстры

```
void WGraph::minpath(int s, double *D, int *P)
{ int i, j, w; double wmin;
  bool *old = new bool[vernum];
  for (i = 0; i < vernum; i++)
  { old[i] = false; D[i] = mat[s][i]; P[i]=s; }
  old[s] = true; D[s] = 0;
  for (i = 1; i < vernum; i++) {
    for (w=-1, wmin=MAX, j=0; j<vernum; j++)
      if (!old[j] && D[j] < wmin)
      { w = j; wmin = D[j]; }
    if (w < 0) break;
    for (old[w] = true, j = 0; j < vernum; j++)
      if (!old[j] && D[j] > D[w]+mat[w][j])
      { D[j] = D[w]+mat[w][j]; P[j] = w; }
  } delete [] old;
}
```

Выделение кратчайшего пути

Оптимальный (кратчайший) путь в вершину j проходит через вершины, для которых оптимальный путь был найден раньше (т.е. ставшие старыми раньше, чем j).

Для каждой вершины существует одна ссылка на непосредственного предка в оптимальном пути – по данным ссылкам можно отследить весь оптимальный путь (назад, вплоть до источника).

Если после выполнения **`minpath`** для вершины **`v`** выполняется **`D[v]=MAX`**, то **`v`** недостижима из **`s`**. В противном случае нужно переходить по ссылкам **`P[v]`**, пока мы не придем в **`s`** (**`P[s]=s`**).

Трудоемкость алгоритма Дейкстры составляет $O(n^2)$.

Вычисление кратчайших расстояний от вершины *s* до всех остальных

Матрица расстояний в общем случае несимметричная

s=0

	0	**1**	**2**	**3**	**4**
0	-	10	5	4	15
1	8	-	2	8	1
2	7	1	-	5	3
3	12	3	13	-	7
4	20	11	9	7	-

\+ вершина 3:

D

0	**1**	**2**	**3**	**4**
0	7	5	***4***	11

\+ вершина 2:

D

0	**1**	**2**	**3**	**4**
0	6	***5***	***4***	8

\+ вершина 1:

D

0	**1**	**2**	**3**	**4**
0	***6***	***5***	***4***	7

Вычисление кратчайших расстояний от вершины *s* до всех остальных

Матрица расстояний:

s=0

	0	1	2	3	4
0	-	10	5	4	15
1	8	-	2	8	1
2	7	1	-	5	3
3	12	3	13	-	7
4	20	11	9	7	-

Кратчайшие расстояния от вершины 0:

D

0	1	2	3	4
0	6	5	4	7

Кратчайшие пути от вершины 0 в обратном порядке:

P

0	1	2	3	4
0	2	0	0	1

Алгоритм Флойда-Уоршалла

Ищет веса кратчайших путей между всеми парами вершин.

Для ориентированного (в общем случае) графа $G = (V, E)$ задана матрица весов дуг $c_{ij} \geq 0,\ i, j = 0 \ldots n-1,\ c_{ii} = 0,$ $c_{ij} = \infty$, если $(i, j) \notin E$.

Рассмотрим последовательность матриц $C^{(0)}, C^{(1)}, \ldots, C^{(n)}$, элементы которых определяются рекуррентно:

$$\left\{ \begin{array}{l} c_{ij}^{(0)} = c_{ij}, \\ c_{ij}^{(l)} = \min\left(c_{ij}^{(l-1)}, c_{il}^{(l-1)} + c_{lj}^{(l-1)}\right),\ l > 0 \end{array} \right\}$$

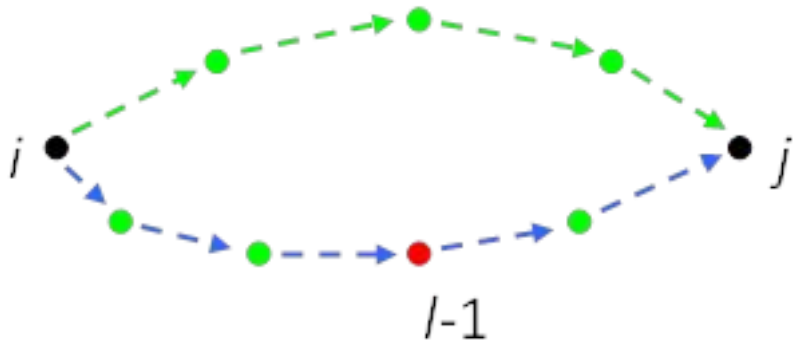
$c_{ij}^{(l)}$ – это вес пути из i в j, в котором в качестве промежуточных могут использоваться только вершины из множества $\{0, 1, \ldots, l-1\}$, которые могут следовать в произвольном порядке и не обязательно все l (число дуг в пути $\leq l+1$).

Алгоритм Флойда-Уоршалла

Докажем, что $c_{ij}^{(l)}$ – вес кратчайшего пути из i в j при заданных ограничениях на промежуточные вершины (по матиндукции):

1. Базис $c_{ij}^{(0)} = c_{ij}$ – очевидно.
2. Пусть $\forall\, i, j \in V$ $c_{ij}^{(l-1)}$ – это вес кратчайшего пути из i в j, проходящего только через вершины $\{0,1,\ldots,l-1\}$ (зеленые).

   На следующем шаге кратчайший путь либо останется прежним ($c_{ij}^{(l)} = c_{ij}^{(l-1)}$), либо обязательно пройдет через вершину $l-1$ (если $c_{ij}^{(l-1)} > c_{il}^{(l-1)} + c_{lj}^{(l-1)}$).



Алгоритм Флойда-Уоршалла

При включении вершины $l-1$ на путях из i в $l-1$ и из $l-1$ в j не может быть совпадающих промежуточных вершин (в противном случае последнее неравенство не выполняется).

Следовательно, $c_{ij}^{(l)}$ - это действительно вес кратчайшего пути, проходящего только через вершины из множества $\{0,1,\dots,l-1\}$.

Таким образом, значения $c_{ij}^{(n)}$ - это искомые веса кратчайших путей.

Алгоритм Флойда-Уоршалла

Вычисляется матрица весов всех кратчайших путей.

```
double** WGraph::allminpath()
{ int i, j, l;
  double **W = new double* [vernum];
  for (i = 0; i < vernum; i++)
    W[i] = new double[vernum];
  for (i = 0; i < vernum; i++)
    for (j = 0; j < vernum; j++)
      W[i][j] = mat[i][[j];
  for (l = 0; l < vernum; l++)
    for (i = 0; i < vernum; i++) {
      if (W[i][l] < MAX)
        for (j = 0; j < vernum; j++)
          W[i][j]=min(W[i][j], W[i][l]+W[l][j]);
  return W;
}
```

Замечания к алгоритму Флойда-Уоршалла

Алгоритм Флойда-Уоршалла –пример использования динамического программирования.

Условия, при которых применимо ДП:

- оптимизационная задача, для которой выполняется свойство оптимальности для подзадач;
- относительно небольшое (полиномиальное от длины входа) число различных подзадач;
- многократное решение одних и тех же подзадач при поиске общего решения на основе полного перебора вариантов – перекрывающиеся подзадачи.

Замечания к алгоритму Флойда-Уоршалла

Порядок решения задачи с помощью ДП:

- описать структуру оптимальных решений задачи и подзадач;
- найти рекуррентное соотношение, связывающее оптимальные параметры\решения подзадач разных уровней;
- двигаясь снизу вверх, от подзадач самого низкого уровня, вычислять их оптимальные параметры\решения только один раз и сохранять результаты в специальной таблице;
- использовать данные из таблицы при поиске оптимального параметра\решения подзадач следующего уровня.

В приведенном алгоритме **`allminpath`** оптимальные решения подзадач разных уровней последовательно вычисляются в матрице **`W`**.

Эйлеровы циклы и пути

Эйлеров цикл в неориентированном графе:

- начинается в произвольной вершине **a**
- проходит по всем ребрам графа по одному разу
- завершается в вершине **a**.

Условия существования эйлерова цикла:

- граф является связным
- степени всех вершин графа (число инцидентных ребер) четные (т.е. если существует ребро, по которому можно прийти в вершину, то всегда найдется ребро, по которому можно выйти).

Если в графе существуют ровно 2 вершины **a** и **b** с нечетными степенями, то можно найти эйлеров путь, который начинается в **a**, проходит по всем ребрам по одному разу и заканчивается в **b**.

Идея алгоритма построения цикла/пути

1. Вычисляются степени всех вершин. Если условия существования цикла/пути не выполняются, то выход.
2. Выбирается произвольная вершина **a** (для цикла) или выделяются 2 вершины с нечетными степенями **a** и **b** (для пути).
3. Строится произвольный начальный (текущий) цикл из **a** или путь из **a** в **b**.
4. Для всех вершин **v** текущего цикла/пути (начиная с **v=a**) проверяется, есть ли еще не пройденные ребра, инцидентные **v**. Если есть, то выделяется побочный цикл, который начинается и заканчивается в **v** и проходит по не проверенным ранее ребрам. Далее побочный цикл включается в текущий непосредственно за вершиной **v**.

Замечания по алгоритму

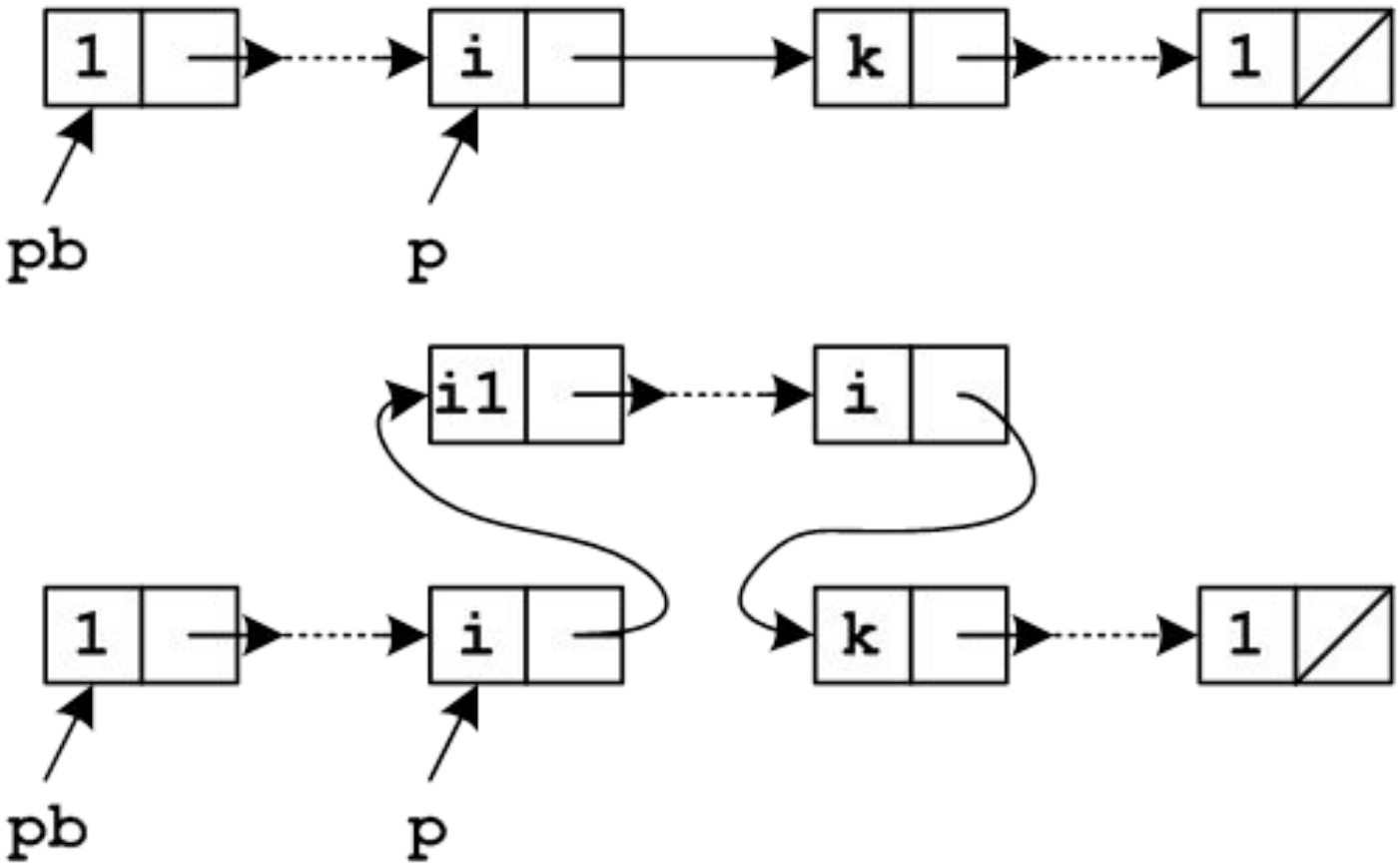
Текущий путь и побочные циклы выгодно строить в виде списков. Используем для этого класс **`List`** из раздела «Линейные списки». Элементы списков будут хранить номера пройденных вершин.

Для представления графа используем класс **`LGraph`** (списки смежных вершин).

Для исключения пройденных ребер можно просто удалять элементы в списках смежных вершин. Чтобы сохранить исходный массив списков **`lst`**, следует создать и модифицировать его копию, но для упрощения алгоритма мы используем сам **`lst`**.

Для вставки побочного цикла в текущий (вставки списка в список с заданной позиции) добавим в класс **`List`** новый метод **`insert(List &sec)`**.

Вставка побочного цикла в текущий



Вставка списка в список

Вставка списка **sec** после текущего элемента (текущая позиция **pcurpos** не изменяется):

```
bool List::insert(List &sec)
{
   if (!pcurpos || sec.is_empty())
      return false;
   (sec.pend)->next = pcurpos->next;
   if (pcurpos == pend) pend = sec.pend;
   pcurpos->next = sec.pbeg;
   sec.pend = sec.pbeg = NULL;
   return true;
}
bool List::is_empty()
{ return (!pbeg)? true : false; }
```

Вспомогательные методы

Проверка существования эйлерова цикла/пути и расчет начальной и конечной вершины (**a** и **b**, для цикла **a=b=0**).

```
bool LGraph::is_euler_path(int &a, int &b)
{ int i, k = 0;
  for (i = 0; i < vernum; i++)
    if (lst[i].getcount() % 2 == 1)
    { k++;
      if (k == 1) a = i;
      else if (k == 2) b = i;
      else return false;
    }
  if (!k) a = b = 0;
  return true;
}
```

Вспомогательные методы

Выделение произвольного пути из **a** в **b** (**a=b** в случае цикла) с исключением просмотренных ребер. Путь формируется в виде списка пройденных вершин **path**.

```
void LGraph::get_path(int a, int b, List &path)
{ int i, vpre = a, vnex;
  path.clear(); path.push_back(a);
  while (true)
  {
    vnex = lst[vpre].pop_front();
    lst[vnex].remove(vpre);
    path.push_back(vnex);
    if (vnex == b) break;
    vpre = vnex;
  }
}
```

Выделение эйлерова цикла/пути

Цикл/путь формируется в виде списка пройденных вершин.

```
List LGraph::get_euler_path()
{
  List curpath, secpath; int beg, end, *pver;
  if (!is_euler_path(beg, end)) return curpath;
  get_path(beg, end, curpath);
  pver = curpath.get_first();
  while (true)
  { if (lst[*pver].is_empty())
    { pver = curpath.get_next();
      if (pver == NULL) return curpath;
      continue;
    }
    get_path(*pver, *pver, secpath);
    secpath.pop_front();
    curpath.insert(secpath);
  }
}
```