图的表示

张晓平

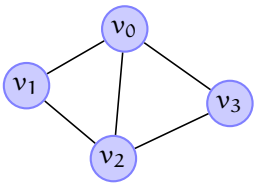
November 22, 2016

1 邻接矩阵

图的邻接矩阵(Adjacency matrix)存储方式是用两个数组来表示图。一个一维数组存储图中顶点信息，一个二维数组(称为邻接矩阵)存储图中的边或弧的信息。

设图G有n个顶点，则邻接矩阵为一个$n \times n$的方阵，定义为

$$e[i][j] = \begin{cases} 1, & \text{if } (v_i, v_j) \in E \text{ or } \langle v_i, v_j \rangle \in E, \\ 0, & \text{otherwise.} \end{cases}$$



$$\begin{array}{c} \\ v_0 \\ v_1 \\ v_2 \\ v_3 \end{array} \begin{array}{c} \begin{array}{cccc} v_0 & v_1 & v_2 & v_3 \end{array} \\ \begin{pmatrix} 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 0 \\ 1 & 1 & 0 & 1 \\ 1 & 0 & 1 & 0 \end{pmatrix} \end{array}$$

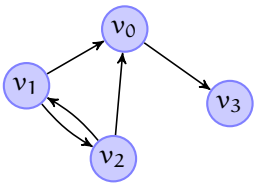
Figure 1: 无向图的邻接矩阵为对称矩阵

有了邻接矩阵，可以很容易地得到无向图中的信息：

1 可以很容易地判定任意两顶点是否有边。

2 顶点v_i的度等于邻接矩阵中第i行的元素之和。

3 欲求顶点v_i的所有邻接点，只需扫描邻接矩阵的第i行，若$e[i][j] = 1$，则v_j为其邻接点。



$$\begin{array}{c} \\ v_0 \\ v_1 \\ v_2 \\ v_3 \end{array} \begin{array}{c} \begin{array}{cccc} v_0 & v_1 & v_2 & v_3 \end{array} \\ \begin{pmatrix} 0 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 \\ 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix} \end{array}$$

Figure 2: 有向图的邻接矩阵不是对称矩阵

有了邻接矩阵，可以很容易地得到图中的信息。

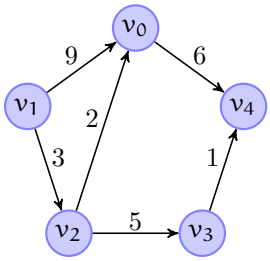
1 顶点v_i的入度等于邻接矩阵中第i列的元素之和；顶点v_i的出度等于邻接矩阵中第i行的元素之和。

2 欲判断顶点v_i到v_j是否有弧，只需查找$e[i][j]$是否为1。

3 欲求顶点v_i的所有邻接点，只需扫描邻接矩阵的第i行，若$e[i][j] = 1$，则v_j为其邻接点。

设图G是网，有n个顶点，则邻接矩阵是一个$n \times n$的方阵，定义为

$$e[i][j] = \begin{cases} w_{ij}, & \text{if } (v_i, v_j) \in E \text{ or } < v_i, v_j > \in E, \\ 0, & \text{if i=j} \\ \infty, & \text{otherwise.} \end{cases}$$



$$\begin{array}{c} \\ v_0 \\ v_1 \\ v_2 \\ v_3 \\ v_4 \end{array} \begin{array}{c} \begin{array}{ccccc} v_0 & v_1 & v_2 & v_3 & v_4 \end{array} \\ \begin{pmatrix} 0 & \infty & \infty & \infty & 6 \\ 9 & 0 & 3 & \infty & \infty \\ 2 & \infty & 0 & 5 & \infty \\ \infty & \infty & \infty & 0 & 1 \\ \infty & \infty & \infty & \infty & 1 \end{pmatrix} \end{array}$$

Figure 3: 网的邻接矩阵

adjmatrix.h

```
typedef char VertexType;
typedef int EdgeType;

#define MAXV 100
#define INF 65535

typedef struct{
  VertexType v[MAXV];
  EdgeType e[MAXV][MAXV];
  int numVertices, numEdges;
} Graph;
```

adjmatrix.h

```
#include "adjmatrix.h"

void CreateGraph(Graph *G)
{
  int i, j, k, w;
  printf("Input␣number␣of␣vertices␣and␣edges:\n");
  scanf("%d␣%d", &G->numVertices, &G->numEdges);
  for(i = 0; i < G->numVertices; i++)
    scanf(&G->v[i]);
  for(i = 0; i < G->numVertices; i++)
    for(j = 0; j < G->numVertices; j++)
      G->e[i][j] = INF;
  for(k = 0; k < G->numEdges; k++) {
    printf("Input␣i,␣j␣and␣weight␣of␣(vi,vj):\n");
    scanf("%d␣%d␣%d", &i, &j, &w);
    G->e[i][j] = w;
    G->e[j][i] = G->e[i][j];
  }
}
```

由代码可知，n个顶点和e条边的无向网络的创建，时间复杂度为$O(n^2+n+e)$，其中邻接矩阵G.e的初始化耗费了$O(n^2)$的时间。

2 邻接表

邻接矩阵对于边数较少的图而言，存在极大的空间浪费。

定义 1. 邻接表是一种数组与链表结合的存储方法。

- 顶点用一维数组存储。数组中每个元素还需存储指向第一个邻接点的指针，以便于查找该顶点的边信息。
- 每个顶点v_i的所有邻接点构成一个线性表。由于邻接点个数不定，故以单链表存储。对无向图，该单链表称为v_i的边表；而对有向图，该单链表则称为v_i作为弧尾的出边表。

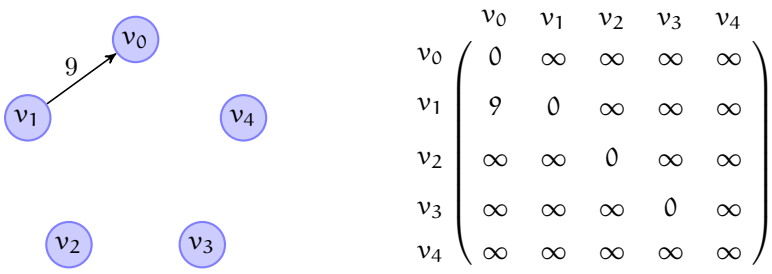


Figure 4: 邻接矩阵不合适的情形

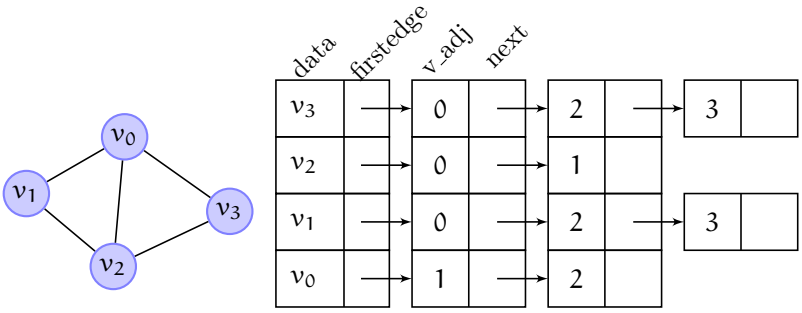


Figure 5: 无向图的邻接表

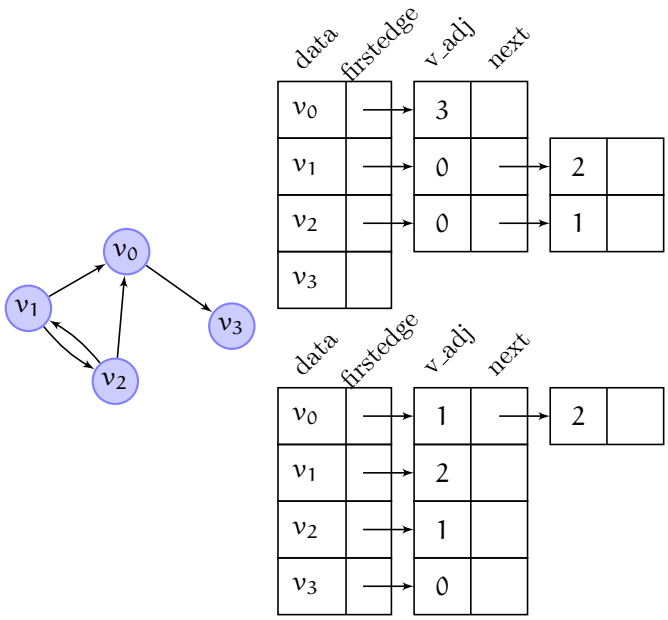


Figure 6: 有向图的邻接表和反向邻接表

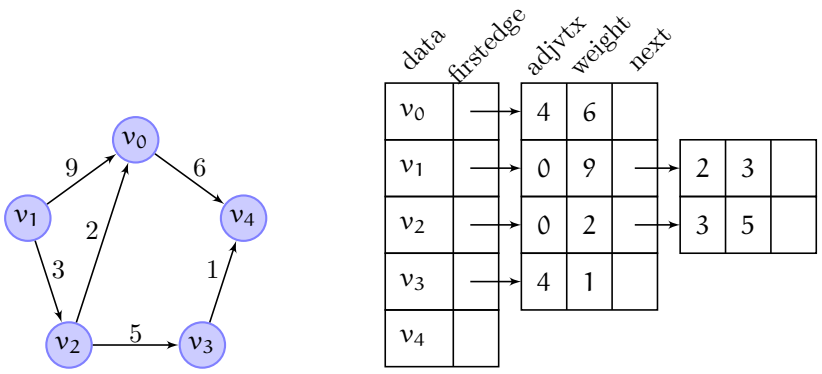


Figure 7: 网络的邻接表

adjlist.h

```
typedef char VertexType;
typedef int EdgeType;
typedef struct EdgeNode {
  int v_adj;
  EdgeType wt;
  struct EdgeNode * next;
} EdgeNode;

typedef struct VertexNode {
  VertexType data;
  EdgeNode * firstedge;
} VertexNode, AdjList[MAX];

typedef struct{
  AdjList adjlist;
  int numVertices, numEdges;
} GraphAdjList;
```

CreateGraph.c

```
#include"adjlist.h"
void CreateGraph(GraphAdjList * G)
{
  int i, j, k;
  EdgeNode * e;

  printf("input␣number␣of␣nodes␣and␣edges:\n");
  scanf("%d,␣%d", &G->numVertices, &G->numEdges);
  for(i = 0; i < G->numVertices; i++) {
    scanf("%d", &G->adjList[i]->data);
    G->adjList[i].firstedge = NULL;
  }

  for(k = 0; k < G->numEdges; k++) {
    printf("input␣index␣of␣vertex␣on␣edge(vi,␣vj):\n");
    scanf("%d,␣%d", &i, &j);
    e = (EdgeNode *) malloc(sizeof(EdgeNode));
    e->v_adj = j;
    e->next = G->adjList[i].firstedge;
    G->adjList[i].firstedge = e;

    e = (EdgeNode *)malloc(sizeof(EdgeNode));
    e->v_adj = i;
    e->next = G->adjList[j].firstedge;
    G->adjList[j].firstedge = e;
  }
```

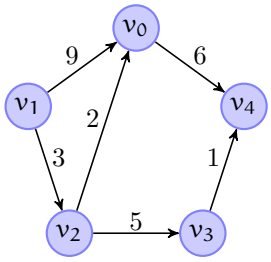
```
}
```

由代码可知，创建n个顶点和e条边的图，时间复杂度为O(n + e)。

3 边集数组

边集数组由两个一维数组构成。一个存储顶点的信息；一个存储边的信息，包括起点下标、终点下标和权重。

边集数组关注的是边的信息，若想在边集数组中查找一个顶点的度需要扫描整个边数组，效率并不高。因此它更适合对各条边依次进行处理的操作， 而不适合与顶点相关的操作。关于边集数组的应用在后面的Kruskal算法中会有介绍。



	v_0	v_1	v_2	v_3	v_4
	begin		end		weight
edge[0]	0		4		6
edge[1]	1		0		9
edge[2]	1		2		3
edge[3]	2		3		5
edge[4]	3		4		1
edge[5]	2		0		2

Figure 8: 边集数组