IPL 电子信息学院 武汉大学 Wuhan University

算法与数据结构（基于现代C++的方法及实践）

ALGORITHM & DATA STRUCTURE IN MODERN C++

第10章 图

王文伟 Wang Wenwei, Dr.-Ing.
Tel: 189-71562600
Email: wwwang@aliyun.com
课程QQ群：珞珈EIS数据结构与算法， 668792335

电子信息学院 Table of Contents 武汉大学 Wuhan University

本章位置

第1章 绪论
第2章 C++编程基础
第3章 遍历、迭代与递归
第4章 字符串
第5章 排序算法
第6章 线性表
第7章 栈与队列
第8章 数组和广义表
第9章 树和二叉树
第10章 图
第11章 查找算法

本章介绍具有非线性关系的图结构，重点讨论图的基本概念及图的存储结构，还将介绍图结构中的常用算法，如遍历算法、图的生成树和最短路径等。

电子信息学院 Table of Contents 武汉大学 Wuhan University

10.0 简介
10.1 图的定义与基本术语
10.2 图的存储结构
10.3 图的遍历
10.4 最小代价生成树
10.5 最短路径

10.0 Introduction

- 图(Graph)是数据元素集合及元素间的关系集合组成的数据结构，元素之间的关系没有限制，任意元素之间可以有关系 (相邻)，即每个数据元素可有多个前驱元素，多个后继元素。图是一种比线性表和树更复杂的非线性数据结构。
- 图是表示离散结构的一种有力的工具，可以用来描述现实世界的众多问题。图论是离散数学的重要分支。
- 本章介绍具有非线性关系的图结构，重点讨论图的概念、存储结构和遍历，并讨论图的生成树、最短路径等。

10.1 图的定义与基本术语

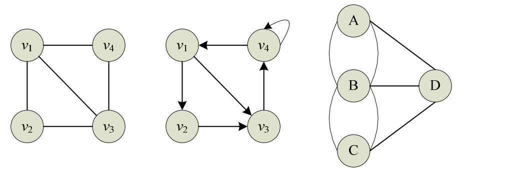
10. 1.1 图的定义
10. 1.2 结点与边的关系
10. 1.3 子图与生成子图
10. 1.4 路径、回路及连通性
10. 1.5 图的基本操作

10.1.1 图的定义

- 图(graph)是由结点集合及结点间的关系集合组成的一种数据结构，任意两个元素之间都可能有某种关系。图中结点（node）之间的关系称为边(edge)。一个图记作G=(V, E)。

V = { x | x∈某个数据元素集合}

E = { $e(x,y)$ | x,y∈V } 或 E = { e<x,y> | x, y∈V }



(a) 无向图G_1　(b) 有向图G_2　(c) Koenigburg七桥图G_3

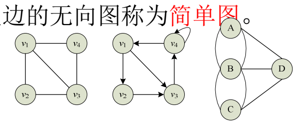
图的术语

◆undirected edge:用结点的无序偶对$e(x, y)$代表一条无向边，相邻关系。undirected graph:无向图

◆directed edge:用结点的有序偶对$e<x, y>$代表一条有向边 (弧， arc)， <x, y>和<y, x>分别表示两条不同的有向边。directed graph:有向图

◆起点和终点是同一个结点的边，即边$e(v, v)$ 或$e<v, v>$，称为环（loop）。例如，G_2中的边$<v_4, v_4>$就是环。

◆无环且无重边的无向图称为简单图。



(a) 无向图G_1　(b) 有向图G_2　(c) Koenigsburg七桥图G_3

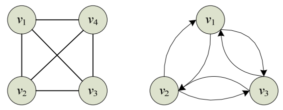
图的术语(II)

◆complete graph:完全图。边数达到最大值，n个结点的完全图记为K_n。有n个结点的无向图，其边的最大数目为$n×(n - 1)/2$；有n个结点的有向图，其弧的最大数目为$n×(n - 1)$。

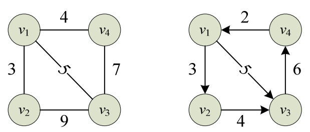
◆sparse graph和dense graph: 有n个结点的图，其边的数目如果远小于n^2，则称为稀疏图。图的边数如果接近最大数目，则称为稠密图。

◆weighted graph或network:带权图或网，每条边上都加注一个称作权（weight）的数值，表示结点对相关的强度信息，例如：从一个结点到另一个结点的距离、花费的代价、所需的时间等。

图的示例



(a) 无向完全图G_4　(b) 有向完全图G_5



(a) 带权的无向图G_6　(b) 带权的有向图G_7

10.1.2 结点与边的关系

◆ adjacent node :若$e(v_i, v_j)$是无向图中的一条边，则称v_i和v_j是相邻结点，边$e(v_i, v_j)$与结点v_i和v_j相关联。

◆ Degree:与结点v相关联的边的数目称为结点的度，表示为TD(v)。度为1的结点称为悬挂点。

◆ 在有向图中，以v为终点的弧数称为v的入度ID(v)；以v为起点的弧数称为v的出度OD(v)。出度为0的结点称为终端结点。TD(v) = ID(v) + OD(v)

◆ 度与边数的关系:

$$\sum_{i=1}^{n} ID(v_i) = \sum_{i=1}^{n} OD(v_i) = e$$ 有向图

$$e = \frac{1}{2}\sum_{i=1}^{n} TD(v_i)$$ 无向图

$$\sum_{i=1}^{n} TD(v_i) = \sum_{i=1}^{n} ID(v_i) + \sum_{i=1}^{n} OD(v_i) = 2e$$

10.1.3 子图与生成子图

◆ subgraph:设图G = (V, E)，G’ = (V’, E’)，若V’≤V，E’≤E，并且E’中的边所关联的结点都在V’中，则称图G’是G的子图。如果G’ ≠ G，则称G’是G的真子图。

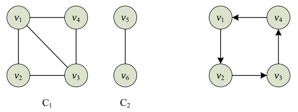
◆ spanning subgraph: 如果G’是G的子图，且V’=V，称图G’是G的生成子图。=> 生成树

10.1.4 路径、回路及连通性

◆ 路径、路径长度、回路:在图中，若从结点v_i出发，沿一些边依次经过结点v_{p1}，v_{p2}，…，v_{pm}到达结点v_j，则称结点序列(v_i，v_{p1}，v_{p2}，…，v_{pm}，v_j)是从v_i到v_j的一条路径。这条路径上边的数目定义为路径长度。如果在一条路径中，除起点和终点外，其他结点都不相同，则称为简单路径。起点和终点相同且长度大于1的简单路径成为回路。带权图中，从起点到终点的路径上各条边的权值之和称为这条路径的（加权）路径长度。

连通图

- connected graph:在无向图G中，若从结点v_i到v_j有一条路径，则称v_i和v_j是连通的。若图G中任意两个结点都连通，则称G为连通图。
- connected component:非连通图的极大连通子图称为该图的连通分量。
- 一个有向图G中，若存在一个结点v_0，从v_0有路径可以到达图G中其他所有结点，则称为有根的图，称v_0为图G的根。



(a) 无向图的两个连通分量C_1和C_2　(b) 强连通有向图G_8

10.1.5 图的基本操作

- Initialize：初始化。建立一个图实例。
- AddNode /AddNodes：在图中设置、添加结点。
- Get/Set：访问。获取或设置图中的指定结点。
- Count：求图的结点个数。
- AddEdge/ AddEdges：在图中设置、添加边，即结点之间的关联。
- Nodes/Edges：获取结点表或边表。
- Remove：删除。从图中删除一个元素及相关联的边。
- Contains/IndexOf：查找。在图中查找满足某种条件的数据元素。
- Traversal: 遍历。按某种次序访问图中的所有结点，并且每个结点恰好访问一次。
- Copy：复制。复制一个图。

10.2 图的存储结构

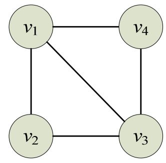
图是结点和边的集合，图的存储结构要记录这两方面的信息。

- 结点的集合可以用一个称之为结点表的线性表来表示；
- 图中一条边表示两个结点的邻接关系，图的边集可以用邻接矩阵（adjacency matrix）或邻接表（adjacency list）表示。邻接矩阵是一种顺序存储结构，而邻接表是一种链式存储结构。
  - 10. 2. 1 图结构的邻接矩阵表示法
  - 10. 2. 2 图结构的邻接表表示法

10.2.1 图结构的邻接矩阵表示法

- 图结构的邻接矩阵用来表示边集，即结点间相邻关系集合。设G=(V, E)是一个具有n个结点的图，G的邻接矩阵A是具有下列性质的n阶方阵：

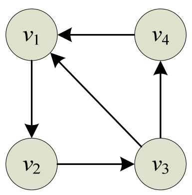
$$a_{ij}=\begin{cases}1 & 若e(v_i,v_j)\in E或\mathbf{e}<v_i,v_j>\in E\\ 0 & 若e(v_i,v_j)\notin E或\mathbf{e}<v_i,v_j>\notin E\end{cases}$$



$$\mathbf{A}_1=\begin{bmatrix}0&1&1&1\\1&0&1&0\\1&1&0&1\\1&0&1&0\end{bmatrix}$$

邻接矩阵

- 无向图的邻接矩阵是对称的，即$a_{ij} = a_{ji}$。有向图的邻接矩阵不一定对称。
- 用邻接矩阵表示一个有n个结点的图结构，需要n^2个存储单元。空间复杂度为$O(n^2)$

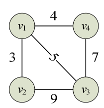


$$\mathbf{A}_8=\begin{bmatrix}0&1&0&0\\0&0&1&0\\1&0&0&1\\1&0&0&0\end{bmatrix}$$

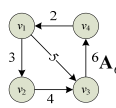
带权图的邻接矩阵

- 在带权图中，设w_{ij}表示边（v_i, v_j）或<v_i, v_j>上的权值，该图的邻接矩阵定义如下：

$$a_{ij}=\begin{cases}w_{ij} & 若v_i\neq v_j且(v_i,v_j)\in E或<v_i,v_j>\in E\\ \infty & 若v_i\neq v_j且(v_i,v_j)\notin E或<v_i,v_j>\notin E\\ 0 & 若v_i=v_j\end{cases}$$



(a) 带权的无向图G_6



(b) 带权的有向图G_7

$$\mathbf{A}_6=\begin{bmatrix}0&3&5&4\\3&0&9&\infty\\5&9&0&7\\4&\infty&7&0\end{bmatrix}\quad \mathbf{A}_7=\begin{bmatrix}0&3&5&\infty\\\infty&0&4&\infty\\\infty&\infty&0&6\\2&\infty&\infty&0\end{bmatrix}$$

邻接矩阵与结点的度

- 用邻接矩阵表示图的边集，容易判定任意两个结点之间是否存在边。
- 用邻接矩阵表示图，可求得各个结点的度。
  - 无向图：邻接矩阵第i行上各元素之和是结点v_i的度。
    $$TD(v_i)=\sum_{j=1}^{n}a_{ij}$$
  - 对于有向图，矩阵第i行上各元素之和是结点v_i的出度，第j列上各数据元素之和是结点v_j的入度。

$$OD(v_i)=\sum_{j=1}^{n}a_{ij},ID(v_j)=\sum_{i=1}^{n}a_{ij}$$

图的顶点类的定义

```
template <typename T> struct Vertex{
  T data;  bool visited;
  Vertex(const T& k, bool v = false):
      data(k), visited(v) { }
  Vertex() : data{}, visited(false) { }
  void show()const{cout<<"-"<<data<<" ->";} };
```

- Vertex结构模板表示图中的顶点，成员data存储顶点的数据，成员visited作为顶点是否被访问过的标志，以后在图的遍历操作中将会用到。

邻接矩阵图类的定义

```
template <typename T> class AdjMatG {
Private: int _count = 0;//图的结点个数
  Vertex<T>* *_pVertexList;//结点表
  int* _pAdjMat;//图的邻接矩阵
  ……
}
```

- AdjMatG类模板表示一个具有若干结点、以邻接矩阵存储的图，将图的邻接矩阵存储在一个一维数组_pAdjMat中，而成员变量_pVertexList保存图的结点表。

邻接矩阵图的基本操作

- 1) 构造函数:使用构造函数创建图对象，构造结点表存储指定的结点序列，构造数组存储指定的邻接矩阵。

```
AdjMatG(int nVertex, const T* pVList, const int* pMat) {
  _count = nVertex;
  _pVertexList = new Vertex<T>*[_count];
  for (int i = 0; i < _count; i++)
    _pVertexList[i]=new Vertex<T>(pVList[i]);
  int n = _count * _count;
  _pAdjMat = new int[n];
  for(int i=0;i<n;i++)_pAdjMat[i]= pMat[i];
}
```


析构函数

```
~AdjMatG() {
  for (int i = 0; i < _count; i++) {
    delete _pVertexList[i];
  }
  delete [] _pVertexList;
  delete [] _pAdjMat;
}
```


2) 返回图的结点数

```
int count() const{
    return _count;
}
```

3) 获取或设置指定结点的值

```
const T& operator [](int i) const {
  if (i >= 0 && i < _count)
    return _pVertexList[i]->data;
  else
    throw out_of_range("Index Out Of Range");
}
T& operator [](int i){
  if (i >= 0 && i < _count)
    return _pVertexList[i]->data;
  else
    throw out_of_range("Index Out Of Range");
}
```


4) 查找具有特定值的元素

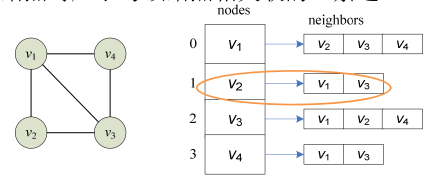
```
int index(const T& k) {
  int j = 0;
  while(j<_count&&k!=_pVertexList[j]->data)j++;
  if (j >= 0 && j < _count)
    return j;
  else return -1;
}
```


10.2.2 图结构的邻接表表示法

- 用邻接矩阵表示图，占用的存储单元个数只与图中结点数有关，而与边的数目无关。一个有n个结点的图需要n^2个存储单元。对于稀疏图，其边数m比n^2少得多，则它的邻接矩阵中就会有很多零元素，造成存储空间上的浪费。
- 这时可用结点表和邻接表来存储图，所占用的存储空间大小既与图中结点数有关，也与边数有关。同是n个结点的图，如果边数$m \ll n^2$，则邻接表表示法需占用的空间比邻接矩阵表示法节省。另外，邻接表保存了与一个结点相邻接的所有结点，这也会给图的操作提供方便。

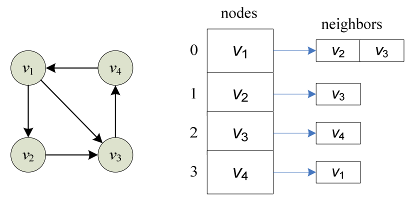
结点表和邻接边表

- 结点表以数组或线性表保存图中的所有结点，其元素的类型是重新定义的图结点类型（GraphNode类），它包括两个基本成员：data和neighbors。data表示结点数据值，neighbors指向结点的**邻接结点表**，简称**邻接表**。
- 邻接表保存与结点相邻接的若干个结点，邻接表中的每个结点对应于与该结点相关联的一条边。



有向图的邻接表

- 无向图的邻接表将每条边的信息存储了两次。因此存储n个结点m条边的无向图占用$n+2m$个结点存储单元。
- 有向图结点的邻接表可以只存储出边相关联的邻接结点，因此，n个结点m条边的有向图的邻接表需要占用$n+m$个结点存储单元。



定义图的结点类型

```
template <typename T> struct GraphNode{
    T data;
    bool visited;
    vector<GraphNode<T>*> *neighbors;
```

- GraphNode结构模板刻画图中的结点，成员data存储结点的数据，成员neighbors存储结点的邻接表，成员visited作为结点是否被访问过的标志。

定义以邻接表存储的图类

```
template <typename T> class Graph {
private:
  vector<GraphNode<T>*> *_pnodes;//结点表
  ……
}
```

- Graph类用来表示一个以邻接表存储的图，其中成员变量_pnodes表示图的结点表，结点表中每个元素对应于图的一个结点，结点类型为GraphNode，每个结点的neighbors成员保存了结点的邻接表。

邻接表图的基本操作

1) 初始化: 使用构造函数创建图对象，存储指定的结点表，并根据给定的邻接矩阵建立邻接表。

```
Graph(int nNodes, const T* pVList, const int* pMat){
  _pnodes = new vector<GraphNode<T>*>();
  for (int i = 0; i < nNodes; i++)
    _pnodes->push_back(new GraphNode<T>(pVList[i]));
  int* p=(int*)pMat;vector<GraphNode<T>*> *pNB;
  for (int i = 0; i < nNodes; i++) {
    pNB = (*_pnodes)[i]->neighbors;
    for(int j=0;j<nNodes;j++)//查找相邻结点j
    if(*p++!=0)pNB->push_back(_pnodes->at(j));
  }  }
```

将邻接矩阵pmat表示的边转换成各结点_pnodes[i]的邻接表neighbors

2) 返回图的结点数

```
int count() const{
    return _pnodes->size();
}
```


3) 获取或设置指定结点的值

```
const T& operator [](int i) const {
  if (i >= 0 && i < count())
    return (*_pnodes)[i]->data;
  else
    throw out_of_range("Index Out Of Range");
}
T& operator [](int i) {
 if (i >= 0 && i < count())
    return (*_pnodes)[i]->data;
  else
    throw out_of_range("Index Out Of Range");
}
```


4) 在图中增加结点

```
void addNode(const T& k) {
  _pnodes->push_back(new GraphNode<T>(k));
}
```


5) 查找具有特定值的元素

```
int index(const T& k) {
  int j = 0;
  while(j<count() && k!=(*_pnodes)[j]->data) j++;
  if (j >= 0 && j < count())
    return j;
  else return -1;
}
GraphNode<T>* findByValue(const T& k) {
  for(auto pnode: *_pnodes)
    if (pnode->data == k)return pnode;
  return nullptr;
}
```

6) 在图中增加边，即增加结点之间的关联

增加一条无向边:

```
void addUndirectedEdge(const T& from, const T& to) {
  GraphNode<T>* fromNode=FindByValue(from);
  if (fromNode == nullptr)return;
  GraphNode<T>* toNode = FindByValue(to);
  if (toNode == nullptr)return;
  fromNode->neighbors->push_back(toNode);
  toNode->neighbors->push_back(fromNode);
}
```

7) 输出各结点的邻接表

```
void show() {
  vector<GraphNode<T>*> *pNB;
  cout << "图的邻接表结构:" << endl;
  for (int i = 0; i < count(); i++) {
    cout << (*_pnodes)[i]->data << " -> ";
    pNB = (*_pnodes)[i]->neighbors;
    for (int j = 0; j < pNB->size(); j++) {
      cout << (*pNB)[j]->data << " + ";
    }
    cout << ".|" << endl;
  }
}
```


10.3 图的遍历

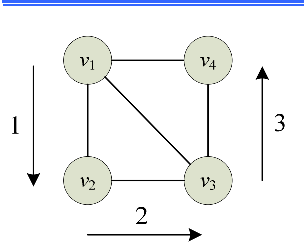
- Traversal操作：从图的一个结点出发，以某种次序访问图中的每个结点，并且每个结点只被访问一次，该过程称为图的遍历。遍历是图的一种基本操作。
- 对于图的遍历，存在两种基本策略：
  - 深度优先搜索DFS遍历：类似于二叉树的先根遍历，depth first search。优先从一条路径向更远处访问图的其他结点，逐渐向所有路径扩展。
  - 广度优先搜索BFS遍历：类似于二叉树的层次遍历，breadth first search。优先考虑直接近邻的结点，逐渐向远处扩展。

10.3.1 基于深度优先策略的遍历

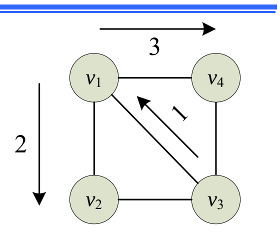
- 图的深度优先搜索(DFS)遍历的基本任务是以深度优先的策略搜索下一个未被访问的结点。递归算法：（为避免同一个结点重复多次访问，在遍历过程中必须对访问过的结点作标记）

1. 从图的一个结点（下标为m，值为s）出发，访问该结点。
2. 查找与结点s相邻且未访问的另一结点（下标为n，值为t）。
3. 若存在这样的结点t，则从t出发继续进行深度优先搜索遍历。
4. 若找不到这样的结点t，说明从s开始能够到达的所有结点都已被访问过，此条路径遍历结束。=> 2

深度优先遍历举例



从顶点v_1出发的一种深度优先遍历序列{v_1, v_2, v_3, v_4}



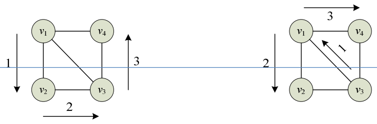
从顶点v_3出发的一种深度优先遍历序列{v_3, v_1, v_2, v_4}

深度优先遍历分析

- 对连通的无向图或强连通的有向图，从某一个结点出发，一次深度优先搜索遍历可以访问图的每个结点；否则，一次深度优先搜索只能访问图中的一个连通分量。
- 设图有n个结点和m条边（$m \geq n$），若用邻接矩阵存储，处理一行的时间为$O(n)$，矩阵共有n行，故时间复杂度为$O(n^2)$。若用邻接表存储，则时间复杂度为$O(n+m)$。

邻接矩阵图的深度优先遍历算法实现 AdjMatG

```
void DepthFirstShow(int m) {
//从结点m :[0 - count-1]开始的深度优先遍历
 _pVertexList[m]->show();
 _pVertexList[m]->visited = true;
 int n=0;int* p=_pAdjMat+m*_count;
 while(n<_count){//查找与m相邻的且未被访问的结点
   if (*(p+n)!=0&&!_pVertexList[n]->visited) {
     DepthFirstShow(n);//递归，继续深度优先遍历
   } n++;
 }
}
```



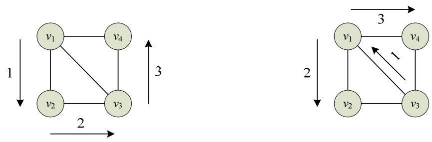
(a) 从顶点v_1出发的一种深度优先遍历序列{v_1, v_2, v_3, v_4} (b) 从顶点v_3出发的一种深度优先遍历序列{v_3, v_1, v_2, v_4}

【例10.1】邻接矩阵图的深度优先遍历算法测试

```
#include "../dsa/AdjMatG.h "
void DepthFirstShowTest(AdjMatG<T>& g) {
  cout << "深度优先遍历:" << endl;
  for (int i = 0; i < g.count(); i++) {
    g.DepthFirstShow(i); cout<<endl;
    g.resetVisitFlag();    }  }
int main (){
  const int CNT = 4;
  string nodes[]={"Vertex1","Vertex2", "Vertex3", "Vertex4"};
  int adjMat[]={0,1,1,1,1,0,1,0,1,1,0,1,1,0,1, 0};
  AdjMatG<string> g(CNT, nodes, adjMat);
  DepthFirstShowTest(g);//BreadFirstShowTest(g);
  return 0; }
```

程序运行结果

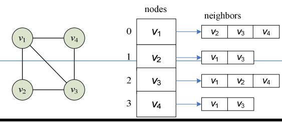
```
深度优先遍历:
-Vertex1 ->-Vertex2 ->-Vertex3 ->-Vertex4 ->
-Vertex2 ->-Vertex1 ->-Vertex3 ->-Vertex4 ->
-Vertex3 ->-Vertex1 ->-Vertex2 ->-Vertex4 ->
-Vertex4 ->-Vertex1 ->-Vertex2 ->-Vertex3 ->
```



(a) 从顶点v_1出发的一种深度优先遍历序列{v_1, v_2, v_3, v_4} (b) 从顶点v_3出发的一种深度优先遍历序列{v_3, v_1, v_2, v_4}

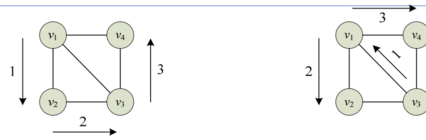
邻接表图的深度优先遍历算法实现 Graph

```
void DepthFirstShow(int m) {
  int i, j;
  vector<GraphNode<T>*> *pNB;
  cout<<"-"<<(*_pnodes)[m]->data<<" ->";
  (*_pnodes)[m]->visited = true;
  pNB = (*_pnodes)[m]->neighbors;
  for (j = 0; j < pNB->size(); j++) {
    if (!(*pNB)[j]->visited) {
      i=index((*pNB)[j]->data);
      DepthFirstShow(i);  //递归访问邻接结点
    }
  } }
```



邻接表图的深度优先遍历算法测试程序运行结果

```
邻接表结构:
Vertex1 -> Vertex2 + Vertex3 + Vertex4 + .
Vertex2 -> Vertex1 + Vertex3 + .
Vertex3 -> Vertex1 + Vertex2 + Vertex4 + .
Vertex4 -> Vertex1 + Vertex3 + .
深度优先遍历:
-Vertex1 ->-Vertex2 ->-Vertex3 ->-Vertex4 ->
-Vertex2 ->-Vertex1 ->-Vertex3 ->-Vertex4 ->
-Vertex3 ->-Vertex1 ->-Vertex2 ->-Vertex4 ->
-Vertex4 ->-Vertex1 ->-Vertex2 ->-Vertex3 ->
```



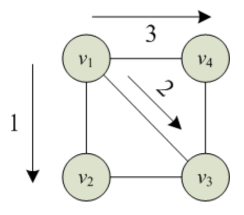
(a) 从顶点v_1出发的一种深度优先遍历序列{v_1, v_2, v_3, v_4} (b) 从顶点v_3出发的一种深度优先遍历序列{v_3, v_1, v_2, v_4}

10.3.2 基于广度优先策略的遍历

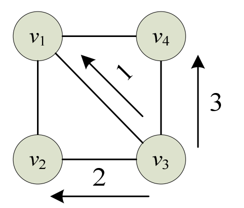
- 图的广度优先搜索(BFS)遍历算法:基本任务是以广度优先的策略搜索下一个未被访问的结点。需设立一个队列来保存访问过的结点，以便在继续遍历中依次访问它们的尚未被访问过的邻接点。

1. 从一个结点（编号为m，值为s）出发，访问该结点。
2. 将访问过的结点（m或s）送入队列（Enqueue）。
3. 当队列不空时，进入以下的循环：
   a) 队首结点（编号为i，值为k）出队（Dequeue）v_i。
   b) 访问与v_i有边相连的且未被访问过的所有结点v_n（编号为n，值为t），访问过的结点v_n入队。
4. 当队列空时，循环结束，说明从结点（m或s）开始能够到达的所有结点都已被访问过。

广度优先搜索遍历举例



从顶点v_1出发的一种广度优先遍历序列{v_1, v_2, v_3, v_4}



从顶点v_3出发的一种广度优先遍历序列{v_3, v_1, v_2, v_4}

广度优先搜索遍历分析

- 由于使用队列保存访问过的结点，若结点v_1在结点v_2之前被访问，则与结点v_1相邻接的结点将会在与结点v_2相邻接的结点之前被访问。
- 如果G是一个连通的无向图或强连通的有向图，从G的任一结点出发，进行一次广度优先搜索便可遍历全图；否则，只能访问图中的一个连通分量。
- 对于有向图，每条弧$<v_i, v_j>$被检测一次，对于无向图，每条边(v_i, v_j)被检测两次。

邻接矩阵图的广度优先遍历算法实现 AdjMatG

```
void BreadthFirstShow(int m) {
  int i,n; queue<int> qi; //设置空队列
  _pVertexList[m]->show();//访问起始结点
  _pVertexList[m]->visited = true;//设置访问标记
  qi.push(m); //访问过的m结点入队
  while(qi.size()!= 0){//队列不空时进入循环
    i=qi.front(); qi.pop();//队首出队，i是结点下标
    n = 0;int* p=_pAdjMat+i*_count;
    while(n<_count){//查找与i相邻且未被访问的结点
      if (*(p+n)!=0 &&!_pVertexList[n]->visited) {
        _pVertexList[n]->show();
        _pVertexList[n]->visited = true;
        qi.push(n);
      } n++;     }    }  }
```


邻接表图的广度优先遍历算法实现 Graph

```
void BreadthFirstShow(int m) {
  vector<GraphNode<T>*> *pNB;
  int i, j;queue<int> qi; //设置空队列
  (*_pnodes)[m]->show(); //访问起始结点
  (*_pnodes)[m]->visited = true; //设置访问标记
  qi.push(m); //访问过的m结点入队
  while(qi.size()!=0) {//队列不空时进入循环
    i=qi.front(); qi.pop(); //队首出队，i是结点下标
    pNB = (*_pnodes)[i]->neighbors;
    for (j = 0; j < pNB->size(); j++) {
      if(!pNB->at(j)->visited){//查找相邻且未被访问的结点
        pNB->at(j)->show();
        pNB->at(j)->visited = true;
        qi.push(index(*(pNB->at(j))));
      }
    } } }
```

10.4 最小代价生成树

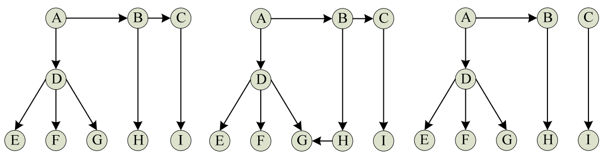
图（graph）可以看成是树（tree）和森林（forest）的推广，树和森林则是图的某种特例，下面首先从图的角度来看待树和森林，然后讨论图的生成树、最小代价生成树等概念。

10.4.1. 树和森林与图的关系
10.4.2. 图的生成树
10.4.3. 图的最小代价生成树

10.4.1 树与图

- 树是一种特殊的图，它是连通的、无回路的无向图。树中的悬挂点称为叶子，其他的结点称为分支点。森林则是诸连通分量均为树的图。
- 树是简单图，因为它无环也无重边。若在树中加上一条边，则形成图中的一条回路；若去掉树中的任意一条边，则树变为森林，整体是非连通图。
- 设图T为一棵树，其结点数为n，边数为m，那么$n-m=1$。

树、森林与图

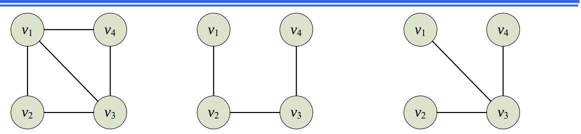


(a) 树是连通图 (b) 树加上一条边成为有回路的图 (c) 树去掉一条边成为森林，非连通图

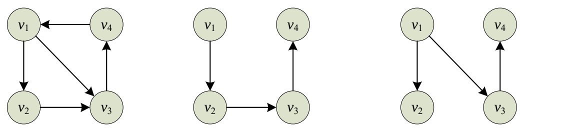
10.4.2 图的生成树

- spanning tree：如果图*T*是（无向）图*G*的生成子图，且*T*是一颗树，则图*T*称为图*G*的生成树。图*G*的生成树*T*包含*G*中的所有结点和尽可能少的边，但仍构成连通图。
- 设G=(V, E)是一个连通的无向图，从*G*的任意一个结点v_0出发进行一次遍历所经过的边的集合为TE，则T=(V, TE)是*G*的一个连通子图，即得到*G*的一棵生成树。任意一个连通图都至少有一棵生成树。生成树不是唯一的。
- 以深度优先遍历图得到的生成树，称为深度优先生成树；以广度优先遍历图得到的生成树，称为广度优先生成树。

图及其生成树

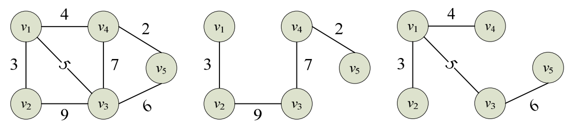


(a) 无向图 (b) 从v_1出发的深度优先生成树 (c) 从v_3出发的广度优先生成树



(a) 有向图 (b) 从v_1出发的深度优先生成树 (c) 从v_1出发的广度优先生成树

带权图的生成树



(a) 带权的无向图 (b) 从v_1出发的深度优先生成树 (c) 从v_1出发的广度优先生成树

- 一个带权图的生成树中，各边的权值之和称为生成树的代价（cost）。一般地，一个连通图的生成树不止一棵，各生成树的代价可能不一样，图中两棵生成树的代价分别为21和18。

10.4.3 图的最小代价生成树

- 设*G*是一个连通的带权图，*w*(*e*)为边*e*上的权，*T*为*G*的生成树，*T*中各边权之和称为生成树*T*的权，也称为生成树的代价（cost）。代价最小的生成树称为最小生成树或最小代价生成树（minimum cost spanning tree，MCST或MST）。

$$w(T) = \sum_{e \in T} w(e)$$

构造最小代价生成树的准则和基本方法

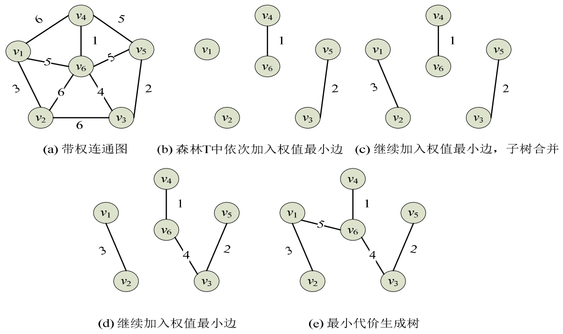
- 最小生成树的4条性质：
  - ➢包含图中的*n*个结点。
  - ➢生成树必须使用且仅使用图中的*n*-1条边。
  - ➢不能使用产生回路的边。
  - ➢最小生成树是权值之和最小的生成树。
- 构造最小代价生成树的基本算法：在逐步求解的过程中利用了最小生成树的一种简称为MST的性质：假设图G=(V, E)是一个连通加权图，若e(u, v)是一条具有最小权值的边，则必存在一颗包含边e(u, v)的最小生成树。

1）Kruskal算法

- 设连通带权图G=(V, E)有n个结点和m条边。
- Kruskal算法的基本思想:
  - 最初先构造一个包括全部n个结点、但无边的森林$T = \{T_1, T_2, \ldots, T_n\}$；依照边的权值大小从小到大将边排序。
  - 然后依次选择权值最小的边，逐边将它们放回到所关联的结点上，但删除会生成回路的边；由于边的加入，使T中的某两棵树合并为一棵，森林T中的树的棵数减1。
  - 经过n - 1步，最终得到一棵有n - 1条边的最小代价生成树。

Kruskal算法

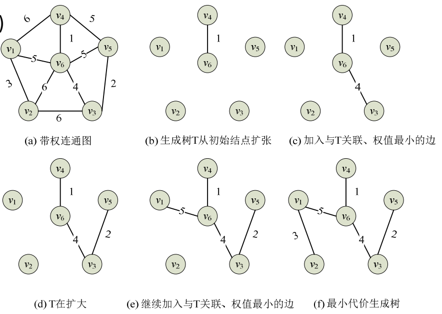
1. 构造n个结点和0条边的森林。
2. 进入循环，依次选择权值最小、但其加入不产生回路的边加入森林，直至该森林变成一棵树为止。



2）Prim算法

- 普里姆算法从连通带权图G的某个结点s逐步扩张成一颗生成树。

1. 生成树T=(U, E_T)开始仅包括初始结点s。
2. 进入循环，选择与T相关联的具有最小权值的边$e(u, v_i)$，$u \in U$，将该边与结点v_i加入到生成树T中，直至产生一个n - 1条边的生成树。

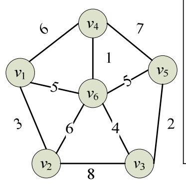


10.5 最短路径

- 图G=(V, E)是一个带权图，从结点u到结点v的一条路径为（$u, v_1, \ldots, v_i, v$），其路径长度不大于从u到v的所有其他路径的路径长度，则该路径是从u到v的最短路径(shortest path)，u称为源点，v称为终点。
- 若给定一个带权图G与源点u，求从u到G中其他结点的最短路径称为单源最短路径问题。
- 所有结点间的最短路径问题：依次将图G中的每个结点作为源点，求每个结点的单源最短路径。

单源最短路径

$$A = \begin{bmatrix} 0 & 3 & \infty & 6 & \infty & 5 \\ 3 & 0 & 8 & \infty & \infty & 6 \\ \infty & 8 & 0 & \infty & 2 & 4 \\ 6 & \infty & \infty & 0 & 7 & 1 \\ \infty & \infty & 2 & 7 & 0 & 5 \\ 5 & 6 & 4 & 1 & 5 & 0 \end{bmatrix}$$



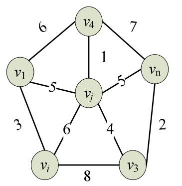
源 点	终 点	路 径	路径长度	最短路径
v_1	v_2	(v_1, v_2)	3	✓
		(v_1, v_6, v_2)	11	
	v_3	(v_1, v_6, v_3)	9	✓
		(v_1, v_2, v_3)	11	
	v_4	(v_1, v_4)	6	✓
		(v_1, v_6, v_4)	6	✓
	v_5	(v_1, v_6, v_5)	10	✓
		(v_1, v_6, v_3, v_5)	11	
	v_6	(v_1, v_6)	5	✓
		(v_1, v_4, v_6)	7	

函数迭代法求解最短路径问题

- 考虑有n个结点的网络，直接用编号1, 2, …, n标识结点，需要求解结点i（i=1, 2, …, n-1）到结点n的最小距离。

$$f(i) = \begin{cases} \min\limits_{1 \le j \le n}\{c_{ij} + f(j)\}, & i = 1, 2, \ldots, n-1 \\ 0, & i = n \end{cases}$$

$f(i)$表示结点i到n的最小距离，$f(j)$表示结点j到n的最小距离，c_{ij}是连接结点i和j之间的距离。含义：为求i到n的最小距离，先对每个结点j，计算i到j的距离c_{ij}，加上j到n的最小距离，结果中值最小的就是i到n的最小距离。



逐步迭代求解

- 迭代的基本思想：先计算各结点经1步（即经过一条边）达到结点n的最短距离$f_1(i)$，再计算各结点经2步到达结点n的最短距离$f_2(i)$，依次类推计算结点i经k步到达结点n的最短距离为$f_k(i)$。具体步骤如下：

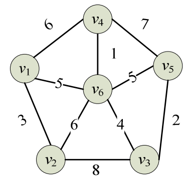
1) 取初始函数$f_1(i)$的值为各结点i经1步达到结点n的距离c_{in}。
2) 对于k=2, 3, …，用下面的方程求$f_k(i)$:

$$f_k(i)=\begin{cases}\min\limits_{1\le j\le n}\{c_{ij}+f_{k-1}(j)\}, & i=1,2,\ldots,n-1\\ 0, & i=n\end{cases}$$

3) 当计算到对所有i=1, 2, …n，均成立$f_{k+1}(i)=f_k(i)$时停止。

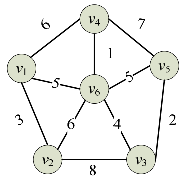
逐步迭代步骤

$$A=\begin{bmatrix}0 & 3 & \infty & 6 & \infty & 5\\ 3 & 0 & 8 & \infty & \infty & 6\\ \infty & 8 & 0 & \infty & 2 & 4\\ 6 & \infty & \infty & 0 & 7 & 1\\ \infty & \infty & 2 & 7 & 0 & 5\\ 5 & 6 & 4 & 1 & 5 & 0\end{bmatrix}$$



$$F_{ki}=\begin{bmatrix}f_1(1)=c_{1n} & f_1(2)=c_{2n} & f_1(i)=c_{in} & 0\\ f_2(1) & f_2(2) & f_2(i) & 0\\ f_k(1) & f_k(2) & f_k(i) & 0\\ f_{k+1}(1) & f_{k+1}(2) & f_{k+1}(i) & 0\end{bmatrix}$$

$$f_k(i)=\begin{cases}\min\limits_{1\le j\le n}\{c_{ij}+f_{k-1}(j)\}, & i=1,2,\ldots,n-1\\ 0, & i=n\end{cases}$$



$$A=\begin{bmatrix}0 & 3 & \infty & 6 & \infty & 5\\ 3 & 0 & 8 & \infty & \infty & 6\\ \infty & 8 & 0 & \infty & 2 & 4\\ 6 & \infty & \infty & 0 & 7 & 1\\ \infty & \infty & 2 & 7 & 0 & 5\\ 5 & 6 & 4 & 1 & 5 & 0\end{bmatrix}$$

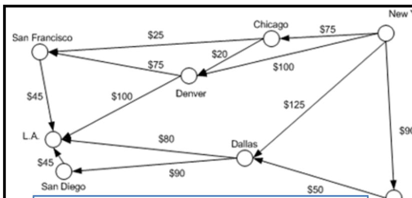
图的邻接矩阵：

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

图中各结点之间的最短路径：

0 3 9 6 10 5
3 0 8 7 10 6
9 8 0 5 2 4
6 7 5 0 6 1
10 10 2 6 0 5
5 6 4 1 5 0

单源最短路径算法 - Dijkstra算法



Distance Table

City	Cheapest Fare
New York	0
Chicago	$75
Miami	$90
Dallas	$125
Denver	$100
San Francisco	Infinity
San Diego	Infinity
L.A.	Infinity

Route Table

City	Route
New York	Null
Chicago	New York
Miami	New York
Dallas	New York
Denver	New York
San Francisco	Null
San Diego	Null
L.A.	Null

Q

Chicago Miami
Dallas Denver San Francisco
San Diego L.A.

Distance Table

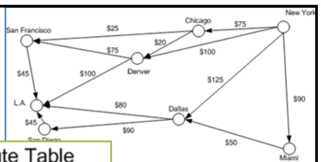
City	Cheapest Fare
New York	0
Chicago	Infinity
Miami	Infinity
Dallas	Infinity
Denver	Infinity
San Francisco	Infinity
San Diego	Infinity
L.A.	Infinity

Route Table

City	Route
New York	Null
Chicago	Null
Miami	Null
Dallas	Null
Denver	Null
San Francisco	Null
San Diego	Null
L.A.	Null

Q

New York Chicago Miami
Dallas Denver San Francisco
San Diego L.A.



Distance Table

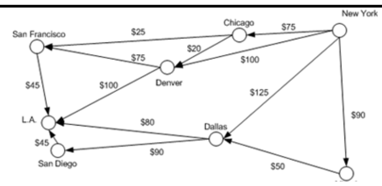
City	Cheapest Fare
New York	0
Chicago	$75
Miami	$90
Dallas	$125
Denver	$95
San Francisco	$100
San Diego	Infinity
L.A.	Infinity

Route Table

City	Route
New York	Null
Chicago	New York
Miami	New York
Dallas	New York
Denver	Chicago
San Francisco	Chicago
San Diego	Null
L.A.	Null

Q

Miami
Dallas Denver San Francisco
San Diego L.A.



Distance Table

City	Cheapest Fare
New York	0
Chicago	$75
Miami	$90
Dallas	$125
Denver	$95
San Francisco	$100
San Diego	$215
L.A.	$145

Route Table

City	Route
New York	Null
Chicago	New York
Miami	New York
Dallas	New York
Denver	Chicago
San Francisco	Chicago
San Diego	Dallas
L.A.	San Francisco

本章学习要点

1. 熟悉图的各种存储结构及其构造算法，了解实际问题的求解效率与采用何种存储结构和算法有密切联系。
2. 熟练掌握图的两种搜索策略的遍历：遍历的逻辑定义、深度优先搜索和广度优先搜索的算法。在学习中应注意图的遍历算法与树的遍历算法之间的类似和差异。
3. 掌握图的最小生成树和最短路径的概念与算法
4. 应用图的遍历算法求解各种简单路径问题。