图

1. 概念

1.1 图的定义

图是由一组顶点和一组能够将两个顶点相连接的边组成。**无向图**则是指边不存在方向，**有向图**是指边存在由一个结点指向另一个节点的方向。

1.2 特殊的图

- 自环
- 含有平行边

由此引申出两种图：

- **多重图**：指的是含有平行边的图。
- **简单图**：指的是不含有自环和平行边的图。

1.3 连通

连通指的是在一个图中，从任意一个节点从存在一条路径可以到达另一个任意的节点，则说明这个图时连通图；**非连通图**是由多个连通的子图构成的，而这些连通子图都是其**极大连通子图**。

1.4 树

树是一幅无环连通图，而互不相连的树组成**森林**。

连通图的**生成树**是它的一个连通子图，图的**生成树森林**是它的所有连通子图的生成树的集合。

当且仅当一幅含有V个节点的图G满足下面五个条件之一的时候，他就是一棵树：

- G有V-1条边并且不含有环；
- G有V-1条边且是连通的；
- G是连通的，但是删掉任何一条边都会使它不再连通；
- G是无环图，但是添加任意一条边都会使它产生一个环；
- G中的任意两个节点之间仅存在一个简单路径。

2. 表示无向图的数据类型

2.1 无向图的API

```
Graph(int V);                                   // 创建一个含有V个节点的无向图，
但不含有边
Graph(In in);                                   // 从标准输入流in读入一幅图
int V();                                        // 返回顶点数
int E();                                        // 返回边数
void addEdge(int v, int w);                     // 向图中增加一条边v-w
Iterable<Integer> adj(int v);                     // 和v相邻的所有顶点
String toString();                              // 图的字符串表示
```

2.2 图的表示方法

包含两个要求：

- 必须为在各种问题中碰到的图预留出足够的空间
- Graph的实例方法实现一定要快

提出三种预设：

1. 邻接矩阵（第一个条件有限制，必须要求V*V大小的矩阵）
2. 边的数组（第二个条件不满足，因为必须遍历所有顶点）
3. 邻接表数组（可以同时满足以上两个条件）

*TIP：平行边的存在排除了邻接矩阵，因为邻接矩阵无法表示平行边，而邻接表数组可以表示，因为数组是一个**包**的类。*

2.3 邻接表数组的实现

邻接表数组是使用一个以顶点为索引的列表数组，其中的每个元素都是与该顶点相邻的顶点的集合，为了表示平行边，我们选择采用**包**的数据类型。

包这种数据类型是指其中的元素可以重复且无序。

下面是采用邻接表数组的代码实现

```
public class Graph {
    private int V;      // 顶点数
    private int E;      // 边数目
    private Bag<Integer>[] adj;     // 邻接表数组

    public Graph (int V) {
        this.V = V;
        this.E = 0;
        this.adj = (Bag<Integer>) new Bag[V];
        for (int v = 0; v < V; v++) {
            adj[v] = new Bag<Integer>[];
        }
    }

    public Graph (In in) {
        this(in.readInt());
        int E = in.readInt();
        for (int i = 0; i < E; i++) {
            // 增加一条边
            int v = in.readInt();
            int w = in.readInt();
            addEdge(v, w);
        }
    }

    public int V () {return V;}
    public int E () {return E;}

    public void addEdge (int v, int w) {
        adj[v].add(w);
```

```
        adj[w].add(v);
        E++;
    }

    public Interable<Integer> adj(int v) {return adj[v];}
};
```

2.4. 任务的完成

我们会讨论很多关于图的任务完成，因此我们设计的目标是要将图的表示和任务的完成要分开来表示。

2.5. 深度优先搜索（DFS）

相当于不断试错的过程，首先选择一条没有到过的节点，并记录当前位置，继续往下寻找，如果找到重复的节点，便回退到之前记录的位置，这样的思路听起来，使用使用到递归的知识。

代码实现

```
public class DepthFirstSearch {
    private boolean[] marked;
    private int count;

    public DepthFirstSearch (Graph G, int s) {
        count = 0;
        marked = new boolean[G.V()];
        dfs(G, s);
    }

    private dfs (Graph G, int v) {
        marked[v] = true;
        count++;
        for (w : G.adj(v)) {
            if (!marked[w]) dfs(G, w);
        }
    }

    public marked (int w) {
        return marked[w];
    }

    public int count () {
        return count;
    }
};
```

2.6. 深度优先寻找单点路径

```
public class DepthFirstPaths {
    private boolean[] marked;
    private int[] edgeTo;      // 记录从起点到该节点连通路径的上一个节点
    private final int s;       // 起点

    public DepthFirstPaths {Graph G, int s} {
```

```
        marked = new boolean[G.V()];
        edgeTo = new int[G.V()];
        this.s = s;
        dfs(G, s);
    }

    private dfs (Graph G, int v) {
        marked[v] = true;
        for (w : G.adj(v)) {
            if (!marked[w]) {
                edgeTo[w] = v;
                dfs(G, w);
            }
        }
    }

    public boolean hasPathTo (int w) {
        return marked[w];
    }

    public Interable<Integer> pathTo (int v) {
        if (!hasPathTo(v)) return null;
        Stack<Integer> path = new Stack<Integer>();
        for (int x = v; x != s; x = edgeTo(x)) {
            stack.push(x);
        }
        stack.push(s);
        return path;
    }
}
```

2.7. 广度优先寻找最短路径

深度优先使用栈的先进后出原则，使用递归这个隐式栈，但是这种方法并没有体现寻找最短路径的方法，与其不同的是广度优先选择使用**先进先出的队列**这一数据结构，保障总是最先到达的点成为路径中的上一个节点，这样的话就可以找到**最短路径**。

实现代码：

```
public class BreadthFirstPaths {
    private boolean[] marked;
    private int[] edgeTo;
    private int s;

    public BreadthFirstPaths (Graph G, int s) {
        marked = new boolean[G.V()];
        edgeTo = new int[G.V()];
        this.s = s;
        bfs(G, s);
    }

    private bfs (Graph G, int s) {
        Queue<Integer> queue = new Queue<Integer>;
        // 对起点进行标记,并且入列
        marked[s] = true;
```

```
        queue.enqueue(s);
        // 进行广度优先查找
        while (!queue.isEmpty()) {
            // 对首先到来的节点进行出列
            int v = queue.dequeue();
            for (int w : G.adj(v)) {
                if (!marked[w]) {
                    marked[w] = true;
                    edgeTo[w] = v;
                    queue.enqueue(w);
                }
            }
        }
    }

    public boolean hasPathTo (int w) {
        return marked[w];
    }

    public Interable<Integer> PathTo (int v) {
        if (!hasPathTo(v)) return null;
        Stack<Integer> path = new Stack<Integer>;
        for (x = v; x != s; x = edgeTo[x]) {
            path.push(x);
        }
        path.push(s);
        return path;
    }
}
```

2.8. 使用深度优先搜索解决连通分量问题

增加一个id[]数组来记录每个节点属于第几个连通分量。count来记录连通分量数量，也用来辅助id[数组的记录。

```
public class CC {
    private int[] id;
    private boolean[] marked;
    private int count;

    public CC (Graph G) {
        marked = new boolean[G.V()];
        id = new int[G.V()];
        for (int s = 0; s < G.V(), s++) {
            if (!marked[s]) {
                dfs(G, s);
                count++;
            }
        }
    }

    private dfs (Graph G, int v) {
        marked[v] = true;
        id[v] = count;
        for (w : G.adj(v)) {
```

```
            if (!marked[w]) dfs(G, w);
        }
    }

    public boolean connected (int v, int w) {
        return id[v] == id[w];
    }

    public int id (int v) {
        return id[v];
    }

    public int count () {
        return count;
    }
};
```

2.9 符号表结合无向图

这种问题节点并不是数字，而是字符串。其他实现并没有区别。

2.10 间隔跳数

其实就是最短路径的求解。

3. 有向图

有向图的实现与无向图的实现类似。

重点是对强连通分量的算法理解

4.最小生成树

加权图是为每条边关联一个权值或是成本的图模型。最小生成树就是生成树中权值之和最小的情况。我们主要有两种算法

- Prim算法
- Kruskal算法

一些需要注意的点：

1. 只考虑连通图
2. 边的权重不一定表示距离
3. 边的权重可能是0或者负数
4. **所有边的权重不同**；如果不同边的权重可以相同那么最小生成树就不一定唯一了。

4.1 Prim算法

它的每一步都会为一颗生长中的树添加一条边。一开始这棵树只有一个顶点，也就是源点，然后在算法实现过程中，不断为他添加V-1条边，每次都选择连接树中的节点和不在树中的节点且权值最小的一条边加入生成树中。 *(贪心算法)*

延时实现和即时实现的区别就是对新加入树中的节点与已在树中的节点的**失效边**的处理时间的区别，**延时处理**，是先将这些失效边留在横切边集合中，等到要删除他们的时候，再检查他们的有效性，而**即时处理**则是直接将失效边从横切边集合中删除。

延时实现:

```
public class LazyPrimMST {
    private boolean[] marked;   // 节点是否在生成树中
    private Queue<Edge> mst;    // 生成树的边
    private MinPQ<Edge> pq;     // 横切边

    public LazyPrimMST (EdgeWeightedGraph G) {
        pq = new MinPQ<Edge>();
        marked = new boolean[G.V()];
        mst = new Queue<Edge>();

        visit(G, 0);
        while (!pq.isEmpty()) {
            // 得到pq中权值最小的边
            Edge e = pq.delMin();

            int v = e.elther(), w = e.other(v);
            if (marked[v] && marked[w]) continue;   // 跳过失效的边
            mst.enqueue(e);     // 将边添加到树中
            if (!marked[v]) visit(G, v);
            if (!marked[w]) visit(G, w);
        }
    }

    private void visit (EdgeWeightedGraph G, int v) {
        // 标记顶点V，并将V所连接的所有未被标记的点的边加入pq中
        marked[v] = true;
        for (Edge e : G.adj(v)) {
            if (!marked[e.other(v)]) {
                pq.insert(e);
            }
        }
    }

    public Iterable<Edge> edges () {
        return mst;
    }
}
```

即时实现:

```
public class PrimMST {
    private Edge[] edgeTo;      // 距离树最近的边
    private double[] distTo;    // distTo[w] = edgeTo[w].weight()
    private boolean[] marked;   // 是否在树中
    private IndexMinPQ<Double> pq;  // 有效横切边

    public PrimMST (EdgeWeightedGraph G) {
        edgeTo = new Edge[G.V()];
        distTo = new double[G.V()];
```

```
        marked = new boolean[G.V()];
        pq = new IndexMinPQ<Double>(G.V());

        for (int i = 0; i < G.V(); i++) {
            distTo[i] = Double.POSTIVE_INFINITY;
        }

        distTo[0] = 0.0;
        pq.insert(0, 0.0);  // 使用顶点0和权值0.0来初始化pq
        while (!pq.isEmpty()) {
            visit(G, pq.delMin());  //将最小的顶点添加到树中
        }
    }

    private void visit (EdgeWeightedGraph G, int v) {
        // 将顶点v加入到树中，并更新数据
        marked[v] = true;
        for (Edge e : G.adj[v]) {
            int w = e.other(v);

            if (marked[w]) continue;  // v-w边失效
            if (e.weight < distTo[w]) {
                // 通过e边w到达树的距离更短，因此更新edgeTo和distTo数组数据
                edgeTo[w] = e;
                distTo[w] = e.weight;
                // 分w点是否在横切边中来
                if (pq.contains(w)) pq.change(w, distTo[w]);
                else pq.insert(w, distTo[w]);
            }
        }
    }
    public Iterable<Edge> edges();
    public double weight();
}
```

5. 最短路径问题

我们在最短路径问题中考虑的是单点最短路径问题，是对加权有向图的讨论

5.1 最短路径树

定义：给定一副加权有向图和一个顶点s，以s为起点的一颗最短路径树是图的一副子图，他包含s和s可达的所有顶点。这棵有向树的根节点是s，树的每条路径都是有向图中的一条最短路径。

通过构造这棵最短路径树，可以提供从s到图中任何顶点的**最短路径**。

5.2 加权有向图的数据结构

- 加权有向边的API

```
public class DirectedEdge {
        DirectedEdge(int v, int w, double weight)
    double weight()
    int from()
    int to()
    String toString()
}
```

- 加权有向图的API

```
public class DirectedEdgeGraph {
    int V();
    int E();
    void addEdge(DirectedEdge e);
    Iterable<DirectedEdge> edges();
    Iterable<DirectedEdge> adj(int v);
}
```

5.3 最短路径的API

对于最短路径的API，设计思路与深度优先和广度优先搜索的思路类似。

```
public class SP {
        SP(DirectedEdgeGraph G, int s);
    double distTo(int v);       //s到v的距离，不存在则为无穷大
    boolean hasPathTo(int v);
    Iterable<DirectedEdge> pathTo(int v);       //从顶点s到v的路径

}
```

5.4 最短路径的数据结构

- **最短路径树中的边**：使用一个由顶点索引的DirectedEdge对象的父链接数组edgeTo[]，其中edgeTo[v]的值为树中连接v和它的父节点的边，也就是s到v的最短路径上的最后一条边。
- **到达起点的距离**：distTo数组，distTo[v]为s到v的已知最短路径的长度。

5.5 边的松弛

我们的最短路径的实现都基于一个操作*（也可以说采用了局部贪心算法的思路）*，这个统一的操作被称为**松弛**，刚一开始，我们只知道图的边和他们的权值，distTo的值除了起点为0其他均被赋值为Double.POSITIVE_INFINITY，随着算法的执行，最短路径的信息就会被存储在distTo和edgeTo数组中。

其中一个重要的操作是边的松弛，定义如下：放松边v->w是指看从s到w的最短路径是否是从s到v，然后再从v到w如果是，则更新数据结构的具体内容。具体操作是比较distTo[w]和distTo[v]+e.weight()的值。如果前者更小，则这条边失效；如果后者更小，则更新数据结构的内容。

5.6 顶点的松弛

事实上，实现顶点的松弛会实现从给定顶点之处的所有边的松弛。

6. Dijkstra算法

算法实现

```
public class Dijkstra {
    private DirectedEdge[] edgeTo;
    private double[] distTo;
    private IndexMinPQ<Double> pq;

    public Dijkstra(EdgeWeightedDigraph G, int s) {
        edgeTo = new DirectedEdge[G.V()];
        distTo = new double[G.V()];
        pq = new IndexMinPQ<Double>(G.V());

        for (int v = 0; v < G.V(); v++) {
            distTo[v] = Double.POSTIVE_INFINITY;
        }
        distTo[s] = 0;

        pq.insert(s, 0.0);
        while (!pq.isEmpty()) {
            relax(G, pq.delMin());
        }
    }

    private void relax (EdgeWeightedDigraphn G, int v) {
        for (DirectedEdge e : G.adi(v)) {
            int w = e.to();
            if (distTo[w] > distTo[v] + e.weight()) {
                distTo[w] = distTo[v] + e.weight();
                edgeTo[w] = e;
                if (pq.contains(w)) pq.change(w, distTo[w]);
                else pq.insert(w, distTo[w]);
            }
        }
    }
}
```