算法设计与分析作业五

作者：吴润泽　　　**学号：**181860109

Email：181860109@smail.nju.edu.cn

2020 年 5 月 5 日

目录

Chapter 10

problem 10.3

起始时刻将 v_1 加入队列中，在第一次循环中将其弹出，v_1 与 n-1 个点均有边相连，故将 n-1 条边及其权值加入队列。在第二次循环时，需要将 n-1 条边中权值最小的边弹出，故需要比较 n-2 次。

problem 10.4

1) 起始 n-1 条边，比较 n-2 次；
第二次 2(n-2) 条边，比较 2n-5 次；
... 因此共需要 $\sum_{i=1}^{n-1}(n-i)i-1=\frac{1}{6}(n^3-7n+6)$ 次比较。

2) 起始时刻将 v_1 加入队列中，在第一次循环中将其弹出，v_1 与 n-1 个点均有边相连，故将 n-1 条边及其权值加入队列。而队列存放的是节点的最小权值信息，最多为 n-1 个。因此此时优先队列存在最多，最多 n-1 个节点。

problem 10.8

1) 最大生成树与最小生成树的本质是相同的，将 prim 算法中维护权值最小的队列改为维护权值最大的队列即可。

```
PrimBig(G):
  initialize all nodes in G as UNSEEN
  initialize the priority queue queNode as empty
  initialize edge set MST as empty /*存放当前局部最大生成树中的边*/
  Select an arbitrary vertex s to start building the maximum spanning tree
  s.candidateEdge:=NULL
  /*每个顶点的candidateEdge,标记它是通过哪条边被连入最大生成树*/
  queNode.INSERT(s,MAX) /*起始点s的权值设为最高，必然先被调度执行*/
  while queNode!=empty:
    v:=queNode.EXTRACT-MAX() /*每次选取队列中权值最大的边*/
    MST:=MST.add{v.candidateEdge} /*将该边加入最大生成树*/
    UPDATE-FRINGE(v,queNode)
  return MST
UPDATE-FRINGE(v,queNode):
```

```
  for neighbor w of v:
    newWeight:=vw.weight
    if w is UNSEEN: /*如果是第一次遍历到*/
      Fringe.INSERT(w,newWeight)  /*将w加入队列*/
    else:
      if newWeight>w.priority:
        w.candidateEdge:=vw
        Fringe.decrease(w,newWeight) /*将w的权值更新为较大的边权*/
```

PrimBig.func

2) 对于最大生成树 T，未被选取的边中，任意选择一条加入 T 中，必定成环，且为环中权值最小的边。即最大生成树未被选择的边即为最小的反馈边集。因此最大生成树算法结束后，遍历所有边记录未被选择的即可。遍历仅需线性时间，最大生成树算法采用优先队列时间复杂度 $O((m+n)\log n)$。

```
PrimFES(G):
  MST:=PrimBig(G)
  return G(E)-MST /*将E中MST抛去剩下的便是FES*/
```

PrimFES.func

problem 10.9

不可能 以结点 s 为起点，使用 Prim 算法构造最小生成树，使用 Dijkstra 算法构造单源最短路。两算法的第一次扩充都是选择 s 相邻结点中边权最小的边。如果在相邻边中的最小边权唯一，则两者选择的边一定相同。

否则假设存在 SU 和 SV 两边权均为最小边权，第一次循环中，Prim 算法选择了 SU，而 Dijkstra 算法选择了 SV。假设 Dijkstra 的结果没有选择 SU，同时因为 S 与 U 有边相连，即 S 到达 U 的路径存在，那么一定是通过 V 或者其他与 S 相邻的结点间接到达 U，而边权值为正，即间接到达 U 的路径一定大于 SU 本身边权。

产生矛盾，故 Dijkstra 一定选择 SU，与 Prim 算法有公共边 SU。

problem 10.11

算法设计 仿照 Prim 算法，对于循环中选择的边 uv，若 $v \in U$ 则不能再以 v 为起点更新优先队列，即 v 只能以某一分支的终点出现。即保证与 U 中结点有关的边只在 T 中出现一次。如果循环结束时，选择的边数不足 n-1 说明不存在最轻生成树。

时间复杂度 判断是否在 U 中，可以采用布尔数组的方式，$O(1)$ 复杂度。整体时间复杂度与 Prim 算法相同，采用优先队列的实现方式，时间复杂度为 $O((m+n)\log n)$。

```
PrimMLT(G,U):
  /*初始化操作*/
  选择不在U中的点s作为Prim起点
  vis:=[] /*维护U中结点被访问的情况*/
  s.candidateEdge:=NULL
  /*每个顶点的candidateEdge,标记它是通过哪条边被连入生成树*/
  queNode.INSERT(s,MIN)
  while queNode!=empty:
    v:=queNode.EXTRACT-MIN()
    MLT:=MLT.add{v.candidateEdge} /*将该边加入生成树*/
    if MLT.size=n-1:return true
    if v not in U: /*在U中的点只能被用来作为边的终点*/
      UPDATE-FRINGE(v,queNode)
  return true
UPDATE-FRINGE(v,queNode):
  for neighbor w of v:
    newWeight:=vw.weight
    if w is UNSEEN: /*如果是第一次遍历到*/
      Fringe.INSERT(w,newWeight)  /*将w加入队列*/
    else if newWeight>w.priority:
        w.candidateEdge:=vw
        Fringe.decrease(w,newWeight) /*更新为较小权值*/
```

PrimMLT.func

problem 10.13

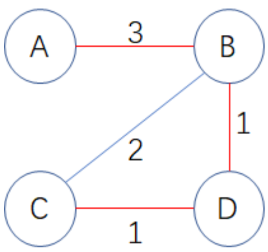
仿照 Kruskal 算法，在算法执行前进行预处理，将 S 中所有边加入 MST 并更新并查集，并在剩余边中排序生成优先队列，进行 Kruskal 算法地执行。

```
KruskalNew(G,U):
  /*初始化操作*/
  initialize the tree MST to S /*将MST初始化为S*/
  initialize the disjoint  with S /*先将S中的边加入并查集*/
  while queNode!=empty:
    vw:=queNode.EXTRACT-MIN()
    if FIND(v)!=FIND(w):
      MST:=MST.add{vw} /*将该边加入生成树*/
      if MST.size=n-1: return /*已经找到n-1条边, 则返回*/
      UNION(v,w) /*将两个并查集合并*/
```

KruskalNew.func

problem 10.15

1) 错误，

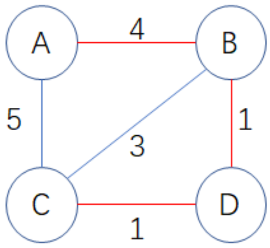


一定包含权值为 2 的唯一最重边。

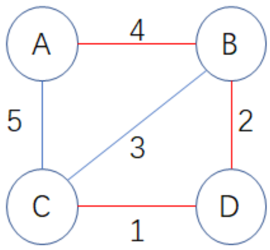
2) 正确，假设 vw 是 G 的唯一最重边，在某个环内。如果 vw 是 MCE，划分将 v、w 划分为两个部分，且 v、w 在环内，则端点分别在两个集合的边不止一条，因为 vw 是 MCE，则它不可能大于其余跨越集合的边的权重，与 vw 是唯一最重边矛盾。因此 vw 一定不是 MCE，而根据课本定理知，MST 中的边一定为 MCE，则 vw 不可能在任何一个 MST 中。

3) 正确，如果 vw 是图 G 唯一最轻边，采用 Dijkstra 算法，则第一步选择全局最小边一定选择 vw。如果 vw 不是唯一最轻边，假设不存在 MST 中含有 vw，将 vw 加入任意一个 MST 中，则必然产生一个环路，从环路中删去任意一条除了 vw 之外的边，形成的生成树权值和一定小于等于 MST，与假设矛盾。因此 vw 必属于某个 MST 上。

4) 正确，与 **3)** 中第一种情况相同。

5) 错误，

，对于环 ABC 其上的 BC 边为唯一最轻边，但最小生成树中不含有边 BC。

6) 错误，

，对于 A 和 C 两节点其最短路径为 AC，但是图中的 MST 唯一且不包含 AC。

7) 正确，对于 Prim 算法，对于边权只涉及到了大小的比较，与正负无关，对于含有负权边算法同样成立。

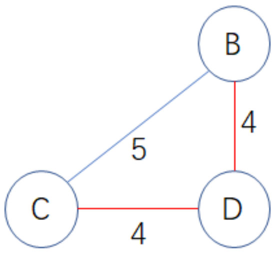
problem 10.16

正确　引理 每条边权值不同的图的最小生成树唯一。
设 G 是所有边权均不相同的无向连通图，设 T1,T2 为 G 的两个 MST，取两 MST 不相同的边集中权值最小的边 e，假设 e 在 T2 而不在 T1 中。将 e 加入 T1，则 T1 中产生一个包含 e 的环。如果 e 不是环中权值最大的边，则违背了最小生成树性质。如果 e 是环中权值最大的边，由于环中不可能均为 T2 中的边，否咋 T2 将产生环，则取该环中不在 T2 的边 v，即 v 也处于两 MST 不相同的边集中，则 v 的权值小于 e，与 e 是环中权值最大的边矛盾。综上每条权值不同的图的最小生成树唯一。
原图和平方图中各边的大小顺序排列不变。我们选取相同的起始点，对两个图执行 Prim 算法，每一轮选择的边均相同，即 T 是 G 的最小生成树当且仅当 T 是 G' 的最小生成树。

problem 10.18

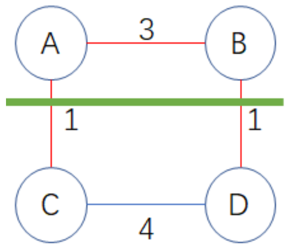
1) problem 10.16 引理已证明。

2)



CD、BD 边权相等，最小生成树即为红边所描。

3) **不等价**



图中最小生成树为红边所连，而将其划分为 AB 和 CD 各一组，跨越两个集合的最小权值边不唯一。

4)

算法设计

1. 在进行 Kruskal 算法合并两个等价类时，分别属于两个等价类的两个点的最长边一定是当前合并的边 (权值按照递增顺序访问)。
2. xy 不在 MST 中，加入 MST 中一定会成环，删去环上除 xy 以外权值最大的边，得到当前树权值和。
3. 枚举所有不在 MST 中的边，得到最小的权值和与 MST 权值和比较，判断是否相等。如果相等，说明存在不止一个 MST。

```
SecMst(G(V,E)):
  mst:=GetMst(G)  /*得到MST的权值和*/
  secmst:=0  /*记录其他生成树最小值*/
  for vw in E:
    if vw not in MST:  /*枚举不在MST中的边*/
      secmst:=min(secmst,mst+weight(w)-path[v][w])
  if secmst=mst:return false  /*MST不唯一*/
  else: return true  /*MST唯一*/
```

```
GetMst(G(V,E)):
  /*初始化操作*/
  sum:=0  /*记录MST的权值和*/
  path:=[][]  /*记录i  j路径上的最长边*/
  while queNode!=empty:
    vw:=queNode.EXTRACT-MIN()
    if FIND(v)!=FIND(w):
      sum:=sum+weight(vw)
      MST:=MST.add{vw} /*将该边加入生成树*/
      for i in V:
        if FIND(i)!=FIND(v): continue
        for j in V:
          if FIND(j)!=FIND(v): continue
          path[i][j]:=path[j][i]:=weight(vw)
          /*边权是递增的，最大值不断更新*/
      if MST.size=n-1: return sum /*已经找到n-1条边，则返回*/
      UNION(v,w) /*将两个并查集合并*/
```

SecMst.func

problem 10.19

1) 假设有用边不在 MST 中，有用边 e 加到 MST 中，因为 e 不属于 G 中的任意环，所以没有环路。此时有 n 条边，与 MST 性质矛盾。故假设不成立，即有用边在最小生成树中。

2) 由problem 10.15 2)可知，环路上的唯一最重边不可能成为 MCE，故不存在于任何 MST 中。

3) 暴力算法

1. 将边权按照降序排列，复杂度 $O(m\log m)$。

2. 依次遍历每条边，判断其是否为危险边，若是则将其删去。

2.1 判断 vw 是否为危险边，先删除 vw，以 w 为起点进行 DFS，记录搜索路径上的最大权，遇到 v 时 DFS 返回。回溯路径上更新最大权值。如果 vw 权值大于最大权值说明是这个环上的危险边，则删去，否则将 vw 加回。时间复杂度为 $O(m(m+n))$。

3. 总的时间复杂度为 $O(m(m+n))$。

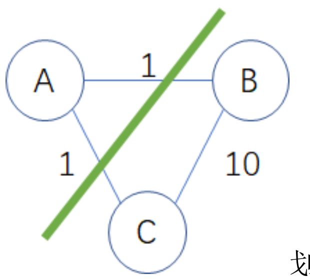
```
AntiKruskal(G):
  降序排列所有边权值，结果存入edges
  for vw in edges:
    G.delete(vw) /*先删除vw，进行判断是否为危险边*/
    if vw.weight<DFSPath(w,v): /*即vw小于环路（或不是环路）的最大权边*/
      G.add(vw) /*不是危险边，将其加回*/
DFSPath(w,v):
  path:=0 /*DFS路径上的最大路径值*/
  w.color:=GRAY
  for neighbor u of w:
    if u.color=WHITE:
      if u=v /*遍历遇到v即环路则返回*/
        path:=wu.weight /*沿原路返回*/
        return path
      else:
        path:= max(wu.weight,DFS(u)) /*回溯后更新path最大值*/
  return path
```

AntiKruskal.func

算法优化 每次循环都要进行一次 DFS 遍历来判断是否为危险边，重复判断过多。由 **2)** 可知，危险边（即不是 MCE）一定不会被选做 MST 中的边，同时由书中定理可知 MCE 一定被 MST 选择，并且权值各不相同，由problem 10.16 引理MST 唯一。因此 MST 未选择的边均为危险边。则采用 Kruskal 算法生成 MST，然后遍历所有边，未被选择的边即为危险边。时间复杂度为 $O(m\log(m))$。

problem 10.21

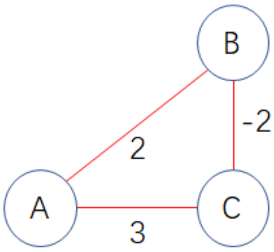
不正确



划分将 A 分为一组，BC 分为一组，分别对两组进行最小生成树，则 BC 边将被选择。两组间的割最小权为 1，则产生的生成树权值和为 11。而最小生成树选择 AB 和 AC 边，权值和为 2。

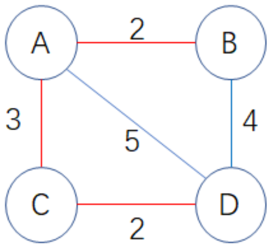
Chapter 11

problem 11.1



以 A 为起点，首先确定到达 B 的最短路径 $A \to B$ 为 2，更新到达 C 最短长度为 0，第二轮确定到 C 的最短路径为 $A \to B \to C$ 为 0。而 A 到达 B 最短路径应为 $A \to C \to B$ 为 1，产生错误。

problem 11.2

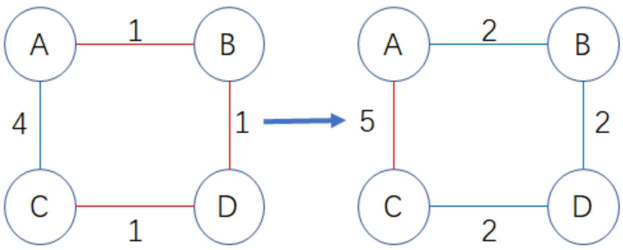
不一定 

，B 到 D 的最短路径为 BD，而最小生成树 (红边所示) 不包含 BD。

problem 11.6

1) 最小生成树不会变化。
原图和边权加 1 后的图中各边的大小顺序排列不变。我们选取相同的起始点，对两个图执行 Prim 算法，每一轮选择的边均相同，因此不会发生变化。

2) 最短路径会发生变化。



AC 的最短路径从 $A \to B \to D \to C$ 边权为 3，变为 $A \to C$ 边权为 5。

problem 11.8

仿照 Dijkstra 算法，将点权转化为边权，对于某一节点更新其邻居 u 的边权时，u 的点权和边权放一起。

```
DijkstraPoint(G,s):
  /*初始化操作*/
  Cost:=[]  /*所有节点的Cost初始化为0*/
  for neighbor w of s:
    queNode.INSERT(s,sw+point[s])  /*到达邻居节点的代价为边权加上s自身点权*/
  while queNode!=empty:
    v:=queNode.EXTRACT-MIN()
    Cost[v]:=v.priority  /*记录v的最短路径权值*/
    UPDATE-FRINGE(v,queNode)
UPDATE-FRINGE(v,queNode):
  for neighbor w of v:
    newWeight:=v.priority+vw.weight+w.priority  /*到达w的代价为到达v的代价加上vw边权和w点权*/
     if w is UNSEEN:  /*如果是第一次遍历到*/
       Fringe.INSERT(w,newWeight)   /*将w加入队列*/
     else if newWeight>w.priority:
          w.candidateEdge:=vw
          Fringe.decrease(w,newWeight)  /*更新为较小权值*/
```

DijkstraPoint.func

problem 11.9

适用 证循环不变式：当结点 u 被加入到集合 S 时，$u.d = \delta(s,u)$ 成立即可。

设结点 u 是第一个加入到集合 S 时是的该不变式不成立的结点，即 $u.d \neq \delta(s,u)$。由于结点 s 是第一个加入到集合 S 中的结点，并且 $s.d = \delta(s,s) = 0$，结点 u 必定与 s 不同。集合 S 在将结点 u 加入之前为非空。此时一定存在某条从 s 到 u 的路径，否则，$u.d = \delta(s,u) = \infty$，与 $u.d \neq \delta(s,u)$ 产生矛盾。故存在一条从 s 到 u 的最短路径 p。设 p 为 $s \to x \to y \to u, x \in S, y \in (V-S)$，

在结点 x 加入集合 S 时，边 xy 被松弛，则有 $y.d = \delta(s,y)$。因为 y 是结点 s 到 u 的一条最短路径上，除了 s 到达邻居节点的边权可能为负外其余节

点为正值。则 $\delta(s,y) \le \delta(s,u)$，即 $y.d = \delta(s,y) \le \delta(s,u) \le u.d$。但是 u 与 y 均在 V-S 里，选择了 u 则有 $u.d \le y.d$。因此 $y.d = \delta(s,y) = \delta(s,u) = u.d$ 与假设矛盾。因此不变式: 当结点 u 被加入到集合 S 时，$u.d = \delta(s,u)$ 依旧成立。

problem 11.10

1) 数学归纳法:

当 n=1，2 时，显然成立。

假设 n=k 时，存在一条哈密顿路径 $v_1 \to v_2 \to \cdots \to v_k$。

当 n=k+1 时，

当 $v_k \to v_{k+1}$ 时，存在哈密顿路径 $v_1 \to v_2 \to \cdots \to v_k \to v_{k+1}$。

当 $v_{k+1} \to v_k$ 时，则按照下标向前寻找，

如果存在 $v_i \to v_{k+1}, v_{k+1} \to v_{i+1}$，

则可构造哈密顿路径 $v_1 \to v_2 \to \cdots \to v_i \to v_{k+1} \to v_{i+1} \to \cdots \to v_k$。

否则，$v_{k+1} \to v_1$ 一定存在，哈密顿路径为 $v_{k+1} \to v_1 \to v_2 \to \cdots \to v_k$。

2) 按照上述逻辑来寻找哈密顿路径。逆向寻找时间复杂度为 $O(V)$，寻找 $V-1$ 条边，则总的时间复杂度为 $O(v^2)$。

```
Hamilton(G(V,E)):
  path:=[]  /*用来存放哈密顿路径的结果*/
  path.push(V(1))   /*确定路径起点*/
  for  i:=1  to  n-1:
    if  map[V(i)][V(i+1)]:  /*如果V(i)指向V(i+1)，则正向加入*/
      path.push(V(i+1))
    else:
      flag:=false  /*是否插入成功*/
      for  j:=i-1  to  1:  /*逆向寻找V(j)指向V(i+1)*/
        if  map[V(j)][V(i+1)]:
          flag:=true
          path.insert(j,V(i+1))  /*在位置j后插入V(i+1)*/
          break
      if  flag=false:  /*没有找到，则将V(i+1)放在头部*/
        path.insert(0,V(i+1))
  return  path
```

Hamilton.func

problem 11.12

1) 在图中删除所有大于 L 的边，然后以 s 为起点进行 DFS，如果可以到达 t，在 DFS 生成树中一定存在路径。时间复杂度为线性。

2) 如果 s 到 t 存在多条路径，每条路径上所需油箱最小容量即为路径上的最大边权值。所有路径中最大边权最小值即为 s 到 t 所需油箱最小容量。

仿照 Dijkstra 算法，在更新结点时维护这条路径上的最大边权。在每次循环选择的边目的地为 t 时，选择对应路径上的最大边权和当前最小容量的较小值即可。

```
DijkstraMaxLeast(G,s):
  /*初始化操作*/
  LeastCost:=Inf /*初始化最小容量为无穷大*/
  for neighbor w of s:
    queNode.INSERT(s,sw) /*到达邻居节点的代价为边权加上s自身点权*/
  while queNode!=empty:
    v:=queNode.EXTRACT-MIN()
    if v=t: /*如果目的地为t*/
      LeastCost:=min(LeastCost,v.priority) /*选取最小的最大权值*/
    UPDATE-FRINGE(v,queNode)
  return LeastCost /*循环结束，返回最小容量即可*/
UPDATE-FRINGE(v,queNode):
  for neighbor w of v:
    newWeight:=max(v.priority,vw.weight) /*维护该队列中的最大权值
    */
    if w is UNSEEN: /*如果是第一次遍历到*/
      Fringe.INSERT(w,newWeight) /*将w加入队列*/
    else if newWeight>w.priority:
        w.candidateEdge:=vw
        Fringe.decrease(w,newWeight) /*更新为较大权值*/
```

DijkstraMaxLeast.func

Chapter 13

problem 13.1

1)

```
FloydNext(G):
  D(0):=W /*初始情况下，两点间的最短路径长度就是它们的边权*/
  for(i:=1 to n):
    for(j:=1 to n):
      if map[i][j]!=Inf: /*如果ij间有边连接，则i的后继即为j*/
        GO[i][j]=j
  for(k:=1 to n):
    for(i:=1 to n):
      for(j:=1 to n):
        if D[i][j]>D[i][k]+D[k][j]:
          GO[i][j]=k /*如果i到j通过k的路径更短，则更新路由表*/
          D[i][j]=D[i][k]+D[k][j]
```

FloydNext.func

2) 算法更新路径 $i \to k \to j$，所选择的 j 的前驱与从选择的节点 k 到节点 j 的一条中间节点取自集合 $\{1, 2, \cdots, k-1\}$ 的最短路径上的前驱是一样的。

```
FloydPrev(G):
  D(0):=W
  for(i:=1 to n):
    for(j:=1 to n):
      if map[i][j]!=Inf: /*如果ij间有边连接，则j的前驱即为i*/
        GO[i][j]:=i
  for(k:=1 to n):
    for(i:=1 to n):
      for(j:=1 to n):
        if D[i][j]>D[i][k]+D[k][j]:
          GO[i][j]:=GO[k][j] /*如果i到j通过k的路径更短，则前缀为
  Back[k][j]*/
          D[i][j]:=D[i][k]+D[k][j]
```

FloydPrev.func

problem 13.2

1) 仿照 Dijkstra 算法，记录每条路径的最小值即可。

```
DijkstraCap(G):
  initialize the priority queue Fringe as empty
  insert some node v to Fringe
  cap:=[] /*cap 原来存放s到达其他点的吞吐量*/
  while Fringe!=empty:
    v:=Fringe.EXTRACT-MIN()
    cap[v]:=v.priority /*存放到达v的结果*/
    UPDATE-FRINGE(v,Fringe)
  return cap
UPDATE-FRINGE(v,Fringe):
  for neighbor w of v:
    w.priority:=min(vw.weight,v.priority)
    /*取路径的最小值，即当前路径已知吞吐量*/
    if w is UNSEEN: /*如果是第一次遍历到*/
      Fringe.INSERT(w,w.priority)  /*将w加入队列*/
    else:
      Fringe.decrease(w,w.priority) /*将w的权值更新为w.priority*/
```

DijkstraCap.func

2) 仿照 Floyd 算法，对路径 $i \to k \to j$ 从 ij，ik，kj 中选取最小权边即可。

```
FloydCap(G):
D(0):=W /*开始的吞吐量记为相连边的边权值*/
for(k:=1 to n):
  for(i:=1 to n):
    for(j:=1 to n):
      D[i][j]:=min(D[i][j],D[i][k],D[k][j]) /*选取3者的最小值，作为最大吞吐量*/
```

FloydCap.func

problem 13.5

1)

必要性 设图 G 的一条欧拉回路为 C。由于 C 经过图 G 的每一条边，而图 G 没有孤立点，所以 C 经过图 G 的每一个顶点，即图 G 为有向强连通图。而对于图 G 的任意一个顶点 v，C 经过 v 时都是从一条边进入，从另一条边离开，因此 C 经过 v 的入边和出边相等，同时 C 不重复地经过了图 G 地每一条边，因此 v 的出度入度相等。

充分性 u 图 G 中每一顶点的入度和出度都相等。在以 s 为起点的路径中，$s \to v_1 \to v_2 \to \cdots \to u$，如果 u 节点不是 s 节点，即 u 出度变为 0，s 的入度变为 0。而节点的入度和出度同时减小，与节点初始入度出度相等矛盾。因此 u 为 s 节点，即回路的最后节点一定为 s。而当沿着路径返回时，如果 v 节点还有出度，则以 v 起点进行深搜，产生子回路。最终遍历所有边形成欧拉回路。

2) 进行 DFS 方法进行遍历，遇到未遍历的边 uv 则以 v 为起点再次进行 DFS 即可。

```
wrapper (G(V,E)):
  vis:=[]
  for edge in E:
    vis[edge]:=0
  DFS(s) /*s为V的某一点*/
DFS (u):
  array:=[]
  for neighbor v of u: /*遍历u的所有邻居*/
    if vis[uv]=0:
      vis[uv]:=1
      DFS(v)
  array.add(u) /*将u加入欧拉回路中*/
  return array
```

Euler.func

problem 13.6

选定一个度为 1 的点作为起点，另一个度为 1 的作为终点，并依次为其编号，起点编号为 1，终点编号为 n。从起点向终点遍历，记录每个点到达起点的距离。任意两个顶点之间的最短距离即为到达起点的距离差。

```
LinearGraph(G):
  dis[1]:=0 /*起点到自身距离为0*/
  for i:=2 to n:
    dis[i]=map[i-1][i]+dis[i-1] /*计算每个点到达起点的距离*/
GetDistance(i,j):
  return i>j?dis[i]-dis[j]:dis[j]-dis[i] /*两点到达起点距离差*/
```

LinearGraph.func

problem 13.7

计算出所有点到达 v_0 的最短路径和 v_0 到达其他顶点的最短路径。则 uv 间的最短路径即为 u 到 v_0 的最短路径加上 v 到 v_0 的最短路径。

```
ShortByV(G,v):
  D:=W /*初始化所有最短路径为起始结点权值*/
  for(k:=1 to n): /*计算所有节点到达v的最短路径*/
    for(i:=1 to n):
      D[i][v]:=min(D[i][v],D[i][k]+D[k][v])
  for(k:=1 to n): /*计算v到达所有节点的最短路径*/
    for(i:=1 to n):
      D[v][i]:=min(D[v][i],D[v][k]+D[k][i])
  for(i:=1 to n):
    for(j:=1 to n):
      D[i][j]:=D[i][v]+D[v][j] /*最短路径为i到v加v到j的最短路径*/
```

ShortByV.func

problem 13.9

对于一个环，如果环中有 i，j，k 三点，则该环的权值和即为 ik 的权值加 jk 的权值加上不经过 k 的 ij 的最短路径。

```
FloydMinLoop(G,v):
  D:=W /*初始化所有最短路径为起始结点权值*/
  MinLoop:=Inf  /*初始化环最小权值为无穷*/
  for(k:=1 to n): /*计算所有节点到达v的最短路径*/
    for(i:=1 to n):
      for(j:=1 to n):
        MinLoop:=min(MinLoop,map[i][k]+D[i][j]+map[k+j]) /*不借助k，i到达j的最短路径*/
    for(i:=1 to n):
      for(j:=1 to n):
        D[i][j]:=min(D[i][j],D[i][k]+D[k][j]) /*确定通过k，i到达j的最短路径*/
  return MinLoop=Inf?NULL:MinLoop /*如果是无穷则返回不存在*/
```

FloydMinLoop.func