6. MST 和 PATH

Problem 6.1

首先需要$n-1$次比较来更新与v_1相连的边的权重，然后将这些边加入到优先级队列中，因为这些边的权重都相同，因此每次插入一条边更新优先级队列需要的比较次数为 1，这样又有了$n-2$次比较（第一个插入的边不需要比较），最后取出最小权值的候选边不需要比较，因此总共需要$2n-3$次比较。

Problem 6.2

1. 若不考虑更新优先级队列需要的比较次数，则在完全图中会进行的边权重的比较次数为：

$$(n-1)+(n-2)+\cdots+1=\frac{n(n-1)}{2}$$

2. 如果v_1与所有节点都相连，且v_1是第一个选取的节点，则会有$n-1$个节点加入优先级队列中。又因为最多可以有$n-1$个节点加入优先级队列，因此优先队列中最多可能存在$n-1$个节点。

Problem 6.3

用堆来实现优先队列，Prim 算法时间复杂度为：

$$\begin{aligned}\mathrm{T}(\mathrm{n},\mathrm{m})&=O\big(n\mathrm{T}(\mathrm{getMin})+n\mathrm{T}(\mathrm{deleteMin})+n\mathrm{T}(\mathrm{insert})+m\mathrm{T}(\mathrm{decreaseKey})\big)\\&=O(n+n\,log\,n+n\,log\,n+m\,log\,n)=O\big((n+m)\log n\big)\end{aligned}$$

1. 在度有界图中，边数$m\leq kn/2$，即$m=O(kn)$，所以 Prim 算法的时间复杂度为：

$$\mathrm{T}(\mathrm{n},\mathrm{m})=O\big((n+m)\,log\,n\big)=O(kn\,log\,n)$$

2. 在平面图中，有$\mathrm{m}\leq 3\mathrm{n}-6$，$\mathrm{d}\leq 2\mathrm{n}-4$，即$\mathrm{m}=\mathrm{O}(\mathrm{n})$，$\mathrm{d}=\mathrm{O}(\mathrm{n})$，所以 Prim 算法的时间复杂度为：

$$\mathrm{T}(\mathrm{n},\mathrm{m})=O\big((n+m)\,log\,n\big)=O(n\,log\,n)$$

Problem 6.4

若优先队列采用数组实现，则 Prim 算法的时间复杂度为$\mathrm{T}(\mathrm{n},\mathrm{m})=O(n^2+m)$；若优先队列采用堆实现，则 Prim 算法的时间复杂度为$\mathrm{T}(\mathrm{n},\mathrm{m})=O\big((n+m)\,log\,n\big)$。

Kruskal 算法的时间复杂度为$\mathrm{T}(\mathrm{n},\mathrm{m})=O(m\log m)$。

在稀疏图中，$\mathrm{m}=\mathrm{O}(\mathrm{n})$，采用堆实现的 Prim 算法和 Kruskal 算法时间复杂度相同，采用数组实现的 Prim 算法时间复杂度比 Kruskal 算法大。

在稠密图中，$\mathrm{m}=\mathrm{O}(\mathrm{n}^2)$，时间复杂度从小到大依次为：采用数组实现的 Prim 算法 < 采用堆实现的 Prim 算法 < Kruskal 算法。

Problem 6.5

1. 最大生成树算法与最小生成树算法原理一样，可以应用贪心策略。
   修改 Kruskal 算法：每次从候选边集中取出权值最大的边，用并查集可以判断加入这条边是否构成环，若不构成环则这条边就是结果的一部分，否则取下一条边继续。
2. 图的最小反馈边集其实是图的最大生成树的互补问题。我们修改上一小题的最大生成树 Kruskal 算法，在该算法每次取权重最大边的过程中，如果判断加入该边会构成环，则将其加入**反馈边集**。算法运行结束，我们可以得到最大生成树，同时也得到了**最小的反馈边集**。

Problem 6.6

不可能。

对于任一顶点 s，我们以 s 为起始点进行 Prim 算法，在 Prim 算法中，第一步会将 s 连接的所有顶点加入优先级队列中，然后从该优先级队列中取出权值最小的边，这条边正是以 s 为起始点进行 Dijkstra 算法选择的第一条边，因为在 Dijkstra 算法的第一步中，算法会选择与 s 相邻的路径（权值）最小的边。

Problem 6.7

1. 不用更新
2. 首先将该边加入 MST，这会在 MST 中构成一个环，我们要在这个环中去掉权值最大的一条边（可能就是新加入

的边)，方法是用一次 DFS 找到这条环上的所有边，然后删除权值最大的边即可，时间复杂度为$O(V+E')$。

3. 不用更新。
4. 我们可以证明，新生成的 MST 中一定包含除该边以外的其它所有边。因此，我们首先将该边从 MST 中去除，然后从剩下可能的候选边中选出权值最小的边即可（可能就是原来的边）。具体方法是：用 DFS 标记出两个连通片，然后我们遍历所有的边，从所有可以连接两个连通片的边中选出权值最小的那个，就构成了新的 MST。时间复杂度为$O(V+E)$。

Problem 6.8

要保证 U 中节点都是叶节点，我们只要保证在挑选边时，这条边不能连接 U 中的两个节点即可。

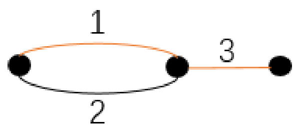
因为要设计$O\big((m+n)\log n\big)$的算法，因此我们需要修改用堆实现优先队列的 Prim 算法：当我们从优先队列中取出一个节点然后更新相邻的节点权值时，如果这个节点是 U 中节点，且相邻节点也是 U 中节点时，则相邻节点的权值保持不变（因为该 MST 中不能包含连接 U 中任意两个节点的边）。

Problem 6.9

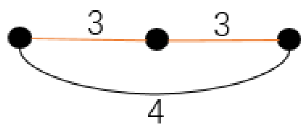
首先将边集 S 中的所有边加入 MST，同时更新并查集，然后将剩余边加入优先队列，在此基础上进行 Kruskal 算法即可。

Problem 6.10

1. 错误，反例如下：红色边为 MST



2. 正确。假设存在某个 MST 包含了最重边e，那么我们一定可以构造出一个比该 MST 权值更小的生成树，方法是将这条环上没被包含在 MST 中的边加入 MST，然后去除最重边e。
3. 错误，因为最轻边可能不唯一，所以如果有两条最轻边构成了一个环，则这两条最轻边不可能都在 MST 中。
4. 正确。考虑 Kruskal 算法过程，这条最轻边一定是第一个被加入 MST 的边。
5. 正确。考虑 Kruskal 算法过程，所有最轻边都不在同一个环中，因此所有最轻边都会被加入 MST。
6. 错误，反例如下：左右两个节点的最短路径不在 MST 中

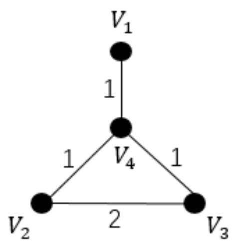


7. 正确。

Problem 6.11

正确。只要图G和G'中每条边权重的顺序相同，那么在 Kruskal 算法中每次挑选出的边的顺序就相同，得到的 MST 就相同。考虑到图 G 中每条边的权重均为正数且各不相同，因为$a<b \Leftrightarrow a^2<b^2\ \ (a,b>0)$，所以，原命题正确。

Problem 6.12

1. 考虑 Kruskal 算法过程可以很容易证明这一点，因为所有边的权重都各不相同，所以 Kruskal 算法在挑选权重最小边时不会产生多个候选边，因此得到的 MST 是唯一的。
2. 

3. 错误，反例如上，划分为两个顶点集合$\{V_1,V_4\}$和$\{V_2,V_3\}$，边V_2V_4和V_3V_4端点分别位于两个集合中，权重最小且不唯一，但是最小生成树是唯一的。

还可以举个反例，一棵边权都为 1 的树本身就是一个 MST，但是显然它不满足第一个条件。

4. 上题等价条件不正确，这题没法做了啊 (╯°Д°)╯︵┻━┻

Problem 6.13

1. 反证法：假设某个有用边不在最小生成树中，因为对于生成树而言，在原图中加上任意一条边都会构成环，因此这个有用边一定会与最小生成树构成环，与有用边的定义矛盾。
2. 反证法：假设 G 的最小生成树中有危险边，则我们一定可以构造出一个权值更小的生成树，方法是：在危险边所在的环中，将不在 MST 中的某个边加入 MST，然后去掉这条危险边。
3. 算法如下：

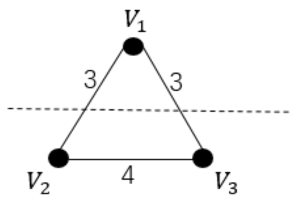
```
algorithm Anti-Kruskal:
    // Finds the minimal spanning tree in graph G = (V, E)
    sort edges of E in descending order
    for i = 1 to m:
        if E[i] forms a loop: // E[i] is dangerous
            G = G - E[i]
```

注意到在上述算法过程中，我们没有直接判断E[i]是否是危险边，而是通过判断E[i]是否构成环路，这两者等价的前提是所有边已经按照递减序排列。

排列所有边需要$O(E\log E)$，判断边是否构成环路可以用 DFS，需要时间为$O\big(E\times(V+E)\big)$，从图中删除一条边需要常数时间，最多删除 E 条边，因此该算法的时间复杂度为$O\big(E(V+E)\big)$。

Problem 6.14

错误。考虑如下反例：



算法将顶点分为上下两部分，在递归求解下半部分时，算法会接着将顶点V_2和V_3分成两部分，然后在组合这两部分时，一定会将边V_2V_3加入 MST，而事实上，该图的 MST 不包含边V_2V_3。因此该算法不正确。

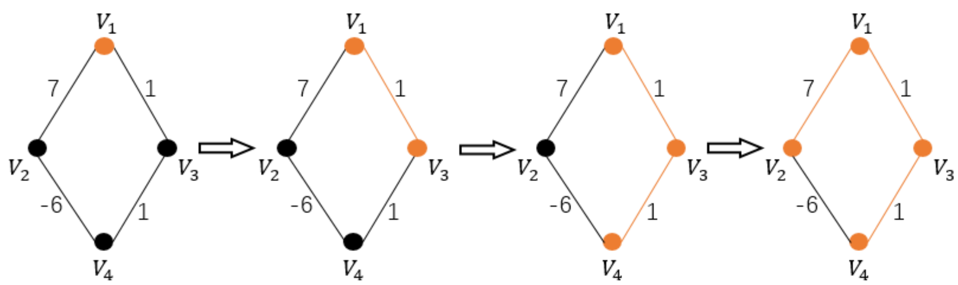
Problem 6.15

1. 首先用 Prim 或者 Kruskal 算法构建一棵最小生成树，然后对于所有不在 MST 中的边进行下列操作：将这条边加入 MST 中，这会形成了一个环，然后从这个环中删除权重最大的边，这就形成了另一棵生成树，这棵树可能是最小生成树，也可能是第二小的生成树，我们记录下新生成的树的权重 w。全部处理完毕以后，我们从中选出权重第二小的，这就是第二小的生成树。

   在已构建的最小生成树中找出一条边所在的环路可以用 LCA（Least Common Ancestor）算法，时间为$O(\log V)$，因此该算法的时间复杂度为$O(T(\mathrm{Prim})/T(\mathrm{Kruskal})+E\log V)$
2. **Amal P M, Ajish K K S. An Algorithm for kth Minimum Spanning Tree[J]. Electronic Notes in Discrete Mathematics, 2016, 53:343-354.**

Problem 6.16

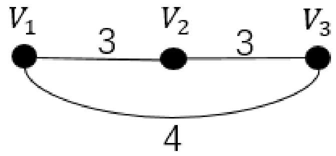
以V_1为起始节点进行 Dijkstra 算法的过程如下：



我们发现 Dijkstra 算法得到的从V_1到V_4的最短路径为$V_1V_3V_4$，长度为 2，而实际上最短路径是$V_1V_2V_4$，长度为 1。

Problem 6.17

不是，反例如下：



从V_1开始运行的 Dijkstra 算法得到的最短路径有V_1V_3和V_1V_2，而图中的最小生成树是V_1V_2和V_2V_3。

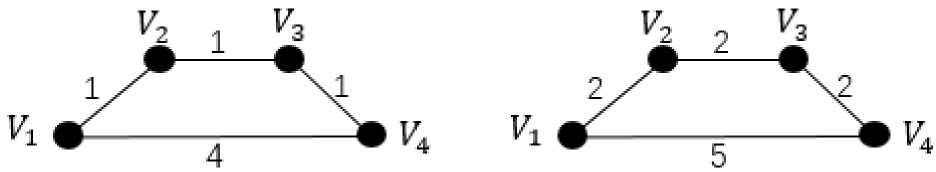
Problem 6.18

在 Dijkstra 算法过程中，当从优先队列中选择一个新的节点 u，然后更新 u 相连的节点权值时，若发现某个相邻节点 v 已经被访问过，且$d(u) + w(u, v) == d(v)$，则存在至少两条路径可以从 s 到达 v。

因此我们可以修改 Dijkstra 算法，首先初始化布尔变量数组$U[1 \dots n]$全部为 TRUE，然后运行 Dijkstra 算法，在算法每次取出一个新结点 u 并更新相邻节点 v 的权值时，若发现$d(u) + w(u, v) == d(v)$，则置$U[v] = FALSE$。

Problem 6.19

1. 最小生成树不会变化。
2. 可能会发生变化。例子如下：



更新权重以前，从V_1到V_4的最短路径是$V_1V_2V_3V_4$，更新权重以后，从V_1到V_4的最短路径是V_1V_4。

Problem 6.20 (推广的最短路径问题)

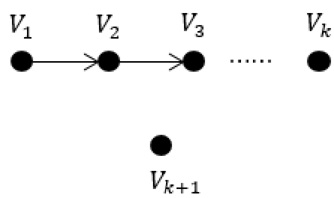
推广的 Dijkstra 算法：更新节点 u 相邻节点 v 的权重公式变为：

$$Cost(v) = min\{Cost(v), Cost(u) + l(u, v) + c(v)\}$$

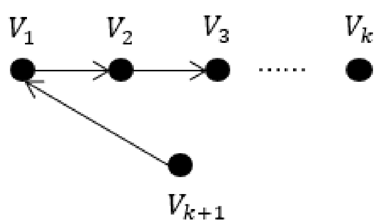
其余过程不变。

Problem 6.22

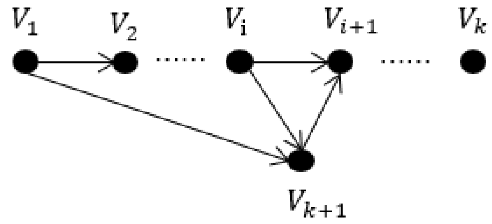
1. 当顶点个数$n = 1$时，图中显然有一条哈密顿路径；
   假设当顶点个数$n = k$时，图中存在一条哈密顿路径；当$n = k + 1$时，我们取其中的 k 个顶点，由归纳假设可知这 k 个顶点之间存在一条哈密顿路径，不妨记为$V_1V_2 \dots V_k$，在竞赛图中，任意两个顶点间都有路径相连，我们考察顶点V_{k+1}与路径$V_1V_2 \dots V_k$的关系，如下图：



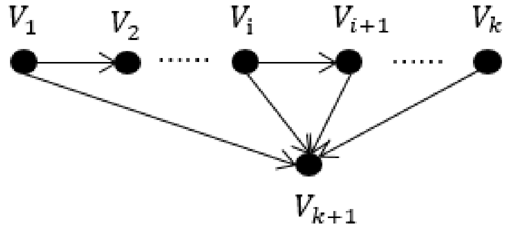
   a. 若存在有向边$V_{k+1}V_1$，如下图所示，则哈密顿路径为$V_{k+1}V_1V_2 \dots V_k$，结果成立。



b. 若存在有向边$\mathrm{V_i}V_{k+1}$和$\mathrm{V_{k+1}}V_{i+1}$，如下图所示，则哈密顿路径为$V_1V_2\ldots V_iV_{k+1}V_{i+1}\ldots V_k$，结果成立。



c. 若上述两种情况都不存在，则我们可以推断出，所有有向边都是从$\mathrm{V_i}(\mathrm{i}=1\ldots\mathrm{k})$指向$\mathrm{V_{k+1}}$，如下图所示，则哈密顿路径为$\mathrm{V_1}V_2\ldots V_k\mathrm{V_{k+1}}$，结果成立。



因此，归纳假设成立，竞赛图中存在哈密顿路径。

2. 应用上述归纳证明的思路，我们可以得到一个递归算法：对于 n 个点的图，如果 n 为 1，则直接返回该点；否则递归计算 n-1 个点的图，得到 n-1 个点的图的哈密顿路径$(\mathrm{v_1},\mathrm{v_2},\ldots,\mathrm{v_{n-1}})$，然后遍历该路径，找出符合归纳法证明时可以成立的点，得到 n 个点的哈密顿路径，然后返回结果。
   该算法时间复杂度的递归式为：
$$\mathrm{T(n)}=\mathrm{T(n-1)}+\mathrm{n}-1$$
   所以，该算法的时间复杂度为$\mathrm{T(n)}=O(n^2)$。

Problem 6.23

1. 设置一个布尔变量数组 reachable[n]，reachable[i]==TRUE 表示从节点 s 到节点 i 可达。我们以节点 s 为起始点运行 DFS 过程，初始置 reachable[s]=TRUE，每当处理节点 u 的相邻节点 v 时，如果 reachable[u]==TRUE 且$l(u,v)\le L$，则置 reachable[v]=TRUE。DFS 结束，通过判断 reachable[t]的值即可判断从 s 到 t 是否存在一条可行路径。
2. 修改 Dijkstra 算法：定义每个节点 u 的权值$\mathrm{d(u)}$表示从 s 到 u 需要的最小油量，每次从优先队列中取出 u，然后更新 u 相邻节点 v 的权值时，有 $\mathrm{d(v)}=\min\{\mathrm{d(v)},\max\{d(u),l(u,v)\}\}$。

Problem 6.24

1. 建立一个无向图，相邻方块为图中相邻节点，然后应用 BFS 算法，遍历到的第一个出口就是最近的出口。
2. 在上题的无向图中去掉所有障碍对应的节点，然后应用 BFS 算法，遍历到的第一个出口就是最近的出口。
3. - 建立一个有向图，边的方向为从左边方块指向右边或者从上边方块指向下边方块，然后应用 Dijkstra 算法，找出从出口到每个节点的最短路径，然后遍历所有出口，找出最近的即可。
   - 和第一小题一样建立一个无向图，然后应用 Dijkstra 算法。
4. 此题无解，前进方向没有约束意味着是无向图，如果还存在负数代价的边，那只要沿着这条边来回走，路径就无穷小了。

Problem 6.25

1. 后继路由表

```
generateRoutingTable(int W[][], int n, int D[][], int GO[][])
    Copy W into D
    for i = 1 to n:
        for j = 1 to n:
            if W[i][j] != infiniy:
                GO[i][j] = j
            else:
```

```
            GO[i][j] = -1
    for k = 1 to n:
        for i = 1 to n:
            for j = 1 to n:
                if D[i][k] + D[k][j] < D[i][j]:
                    D[i][j] = D[i][k] + D[k][j]
                    GO[i][j] = GO[i][k]
```

2. 前驱路由表

```
generateRoutingTable(int W[][], int n, int D[][], int GO[][])
    Copy W into D
    for i = 1 to n:
        for j = 1 to n:
            if W[i][j] != infiniy:
                GO[i][j] = i
            else:
                GO[i][j] = -1
    for k = 1 to n:
        for i = 1 to n:
            for j = 1 to n:
                if D[i][k] + D[k][j] < D[i][j]:
                    D[i][j] = D[i][k] + D[k][j]
                    GO[i][j] = GO[k][j]
```

Problem 6.26

1. 修改 Dijkstra 算法：记每个节点的权值为$\mathrm{cap}(\mathrm{s},\mathrm{u})$，表示从 s 到 u 的所有路径的最大吞吐率，每次从优先队列中取出 u，然后更新 u 相邻节点 v 的权值时，有:
$$\mathrm{cap}(\mathrm{s},\mathrm{v}) = \max\{\mathrm{cap}(\mathrm{s},\mathrm{v}), \min\{cap(s,u), c(u,v)\}\}$$
算法运行结束即可得到结果。
2. 修改 Floyd-Warshall 算法：初始化管道容量数组$\mathrm{c[i][j]}$时，若管道 i 和 j 连通，则$\mathrm{c[i][j]}$为容量值；若管道 i 和 j 不连通，则$\mathrm{c[i][j]} = 0$；若$\mathrm{i} == \mathrm{j}$，则$\mathrm{c[i][i]} = +\infty$。

```
getMaxCapacity(int c[][], int n, int cap[][])
    Copy c[][] into cap[][]
    for k = 1 to n:
        for i = 1 to n:
            for j = 1 to n:
                cap[i][j] = max(cap[i][j], min(cap[i][k], cap[k][j]))
```

Problem 6.27

首先在原图的基础上添加一个节点 s，并且添加从 s 到点集 S 中每个节点的一条边，边的长度设为 0。接下来以 s 为源节点运行 Dijkstra 算法，我们就得到了从 s 到点集 V 中每一节点的最短路径值$\mathrm{d[v]}$，其中每个最短路径值其实就是 S 中某个节点到 v 的最短路径值，因为从 s 到 S 中节点的长度为 0。下面只要求$\min\{\mathrm{d[v]} | \mathrm{v} \in \mathrm{V}\}$即可。算法时间复杂度为$\mathrm{O}\big((\mathrm{m}+\mathrm{n})\log n\big)$，如果是稀疏图，则时间复杂度为$\mathrm{O}(\mathrm{m}\log n)$。

Problem 6.28

从顶点 u 到 v 经过v_0的最短路径为：从 u 到v_0的最短路径加上从v_0到 v 的最短路径。

因此问题可以转换为：找出所有以v_0为源节点的最短路径，再找出所有以v_0为目的节点的最短路径。前一个小问题可用 Dijkstra 算法求得，后一个小问题也可以用 Dijkstra 算法来求，只要事先对图G进行一次翻转操作变为G^{T}。翻转操作的时间复杂度为$\mathrm{O}(\mathrm{m}+\mathrm{n})$，再加上 Dijkstra 算法的代价，该算法的时间复杂度为$\mathrm{O}\big((\mathrm{m}+\mathrm{n})\log n\big)$。

Problem 6.29

先运行 Floyd-Warshall 算法，得到所有点对之间的最短距离矩阵$d[][]$；

然后遍历这个矩阵，寻找$d(u,v)+d(v,u)$的最小值就是答案；

```
call Floyd-Warshall algorithm to compute d[][]
answer = infinity
for u = 1 to n:
    for v = 1 to n:
        answer = min(answer, d[u][v] + d[v][u])
```

Problem 6.30

修改 Bellman-Ford 算法，进行 k 次迭代即可。伪代码如下：

```
Bellman-Ford(Graph G, int w[][], int d[], int parent[])
    for each vertex v in G.V:
        d[v] = infinity
        parent[v] = -1
    d[u] = 0
    for i = 1 to k:
        for each edge(u,v) in G.E:
            Relax(u, v, w)

Relax(int u, int v, int w[][]):
    if d[v] > d[u] + w[u][v]:
        d[v] = d[u] + w[u][v]
        parent[v] = u
```

Problem 6.31

在 Dijkstra 算法的基础上，再设置一个数组$count[1 \dots n]$，其中$count[v]$表示从 u 到 v 的最短路径的数目。初始时置$count[1 \dots n]$为 0，$count[u]=1$，然后从 u 开始运行 Dijkstra 算法，每次取出优先队列的节点 u 时，我们判断节点 u 相邻的节点 v 的最短路径值，如果$d[v]<d[u]+w(u,v)$，则$count[v]=1$；如果$d[v]==d[u]+w(u,v)$，则$count[v]=count[u]+count[v]$。

算法伪代码如下：

```
initialize the priority-queue pq as empty
initialize the array d[1..n] as infinity
initialize the array count[1..n] as empty
d[u] = 0
count[u] = 1
pq.insert(u)
while (pq is not empty):
    u = pq.getMin()
    pq.deleteMin()
    for all vertices v adjcent to u:
        newWeight = d[u] + w(u, v)
        if v.status is unseen:
            d[v] = newWeight
            count[v] = 1
            pq.insert(v)
        else if newWeight < d[v]:
            pq.decreaseKey(d[v], newWeight)
```

```
            d[v] = newWeight
            count[v] = 1
        else if d[u] + w(u, v) == d[v]:
            count[v] = count[v] + count[u]
```

姓名：陆依鸣
学号：151220066
邮箱：luyimingchn@gmail.com