第八次实验报告：贪心算法

陈潇涵 PB13000689

2016 年 5 月 5 日

实验内容

本次实验通过实现最短路径的 Dijkstra 算法和最小生成树的 Prim 算法和 Kruskal 算法，来加深对贪心算法的理解。

实验原理

单源最短路径的 Dijkstra 算法

单源最短路径的 Dijkstra 算法中有两个集合，一个是已经求得最短路径的顶点集合 S，另一个是还需求解最短路径的顶点集合 Q。在每一步选择，都是在第二个集合中选择路径最短的一个，加到集合 S 中，并从集合 Q 中去掉。然后在以这一个点为依据更新第二个集合中的所有点的最短距离。直到第二个集合为空。

正确性：可以用反证法证明对从起点 s 到终点 t 的最短路径上的每个顶点 v，该路径都是从 s 到 v 的最短路径。否则，我们可以找到更短的一条从 s 到 v 的路径，再加上从 v 到 t 的路径，就得到了从 s、到 t 的一条更短路径。

贪心法则的正确性：假设集合 Q 中的距离最短的点为 u，最短距离为 d，即，从源点 s 出发经过集合 S 到顶点 u 的最短距离为 d。而从源点 s 到 Q 中其它点 v 的距离都大于 d，因此每一条经过 v 再到达 u 的路径长度都大于 d。因此 d 一定是从 s 到达 u 的最短距离，此时选择 u 加入集合 S 一定是安全的。

MST 的 Kruskal 算法

Kruskal 算法我觉得还是很好理解的：我们将所有顶点初始化为一个只包含自己的集合，再将所有边按照权重由低到高的顺序进行排序。从排好序的边中我们一条条取出来，如果这条边连接的两个顶点所在的集合相同，即表示这两个顶点已经是连通的了，那么我们不做任何操作；如果这条边的两个端点所在的集合不同，则说明这两个顶点不连通，于是我们将这条边加入到最小生成树的边集合中。重复以上过程知道边集合的大小等于顶点集大小减一。

正确性：然而严格的正确性听韩路新说要用拟阵什么的去证明，于是我一脸懵逼。因为在我看来这个方法的正确性是不言自明的……

时间复杂度：Kruskal 算法的时间复杂度取决于连通子集的数据结构。如果用书上 21 章第 3 节的高级方法，|V| 个顶点的 Make-Set 方法和 |E| 次循环中的 Find-Set 和 Union 方法，根据 21.3 中的结论可以得到总的时间复杂度为 $O((E+V)\alpha(V))$，其中 $\alpha(V)$ 是一个随 V 增长非常缓慢的函数。再考虑一下图的连通性，有 $E \geq V-1$ 以及 $E \leq V^2$，我们有总的时间复杂度为 $O(E\log V)$。但是我用 python 写的时候用了一些 python 的特性，同样能达到上述复杂度，我会在后面实验内容中再详细说明。

MST 的 Prim 算法

Prim 算法也很好理解。Prim 算法中维持两个集合和一些信息：第一个集合 S 是已经确定加入最小生成树的顶点集合，第二个集合 Q 是还需确认的顶点集合，算法执行过程中我们还需要维护 Q 集合中的顶点 u 的信息——u 到集合 S 的距离。

贪心法则：每一次我们都选择到集合 S 距离最短的点 u 加入集合 S，再从集合 Q 中删除。之后我们再更新 Q 中顶点到集合 S 的距离。重复上述步骤直到集合 Q 为空。

正确性：假设某一步贪心选择中选取的边不是任何一棵最小生成树的边。假设最终生成的最小生成树中集合 S 通过路径 $S-u_1-u_2-\ldots-u_m-u$ 使得集合 S 与顶点 u 连通，且 $u_1, u_2, \ldots, u_t$ 都是集合 Q 中的点，由于 u 是距离集合 S 最近的点，我们将边 $S-u_1$ 替换为边 $S-u$，这样我们得到的仍然是一棵生成树，且这棵生成树的代价比原来的生成树小，这与原来的生成树是最小生成树的假设想矛盾。Prim 算法的正确性由此得证。

时间复杂度：Prim 算法的时间复杂度决定于集合 Q 中最小距离顶点选取的实现方式。如果我们用堆排序中的方法将集合 Q 维持成最小堆的形态，然后在每一次选取 u 的时候选取根即可，从 u 中删除点时只需要将根与对中最后一个元素互换，然后维持一下堆的性质，这个操作的时间复杂度是 $\log V$。在初始化步骤我们需要建立最小堆，这里的时间复杂度是 $O(V)$。Prim 算法中的循环部分需要执行 V 次，因此提取最小元素部分的操作的时间复杂

度共需要 $O(V \log V)$。Prim 算法中，更新距离的操作的执行次数之多不超过 $2E$ 次，每一次更新操作中，判断点在 Q 中可以在常数时间内完成，如果我们开辟一个数组来额外记录每个顶点是否在 Q 中即可；修改距离后，我们还需要维持最小堆性质，时间复杂度为 $\log V$。因此更新距离的总时间复杂度为 $O(E \log V)$。因此，Prim 算法的总时间复杂度为

$$O(V \log V) + O(E \log V) = O(E \log V)).$$

实验中出现的问题

实验中没有出现什么问题。

实验内容

```
def dijkstra(G,n,s):
    dist = [ np.inf for i in xrange(n)]
    prec = [ None   for i in xrange(n)]
    visited = []
    dist[s] = 0.0
    while len(visited) != n:
        min_dist = np.inf
        for i in xrange(n):
            if i not in visited:
                if dist[i] < min_dist:
                    min_dist = dist[i]
                    u = i
        visited.append(u)
        for v in xrange(n):
            if v not in visited and G[u,v] + dist[u] < dist[v]:
                dist[v] = G[u,v] + dist[u]
                prec[v] = u
```

图 1: Dijkstra 算法代码截图

```python
def kruskal(G):
    n    = len(G)
    sets = [[i] for i in xrange(n)]
    ans  = []
    edge = []
    for i in xrange(n):
        for j in xrange(i+1,n):
            edge.append({"w":G[i,j], "uv":(i,j)})
    edge.sort(key=lambda e: e["w"])
    for e in edge:
        if len(ans) is n-1:
            return ans
        u,v = e["uv"]
        if sets[u] != sets[v]:
            # if there are more elements in sets[v], exchange u,v
            # this is to cut the expense of unite two sets
            if len(sets[u]) < len(sets[v]):
                u,v = v,u
            for w in sets[v]:
                sets[u].append(w)
            for w in sets[v]:
                sets[w] = sets[u]
            ans.append((u,v))
    return "NO MST!"
```

图 2: Kruskal 算法代码截图

```python
def prim(G):
    n = len(G)
    dist = [np.inf for i in xrange(n)]
    prec = [None   for i in xrange(n)]
    ans   = []
    dist[0] = 0.0
    visited = []
    while len(visited) < n:
        min_dist = np.inf
        for v in xrange(n):
            if v not in visited:
                if dist[v] < min_dist:
                    min_dist = dist[v]
                    u = v
        visited.append(u)
        for v in xrange(n):
            if v not in visited and G[u,v] < dist[v]:
                dist[v] = G[u,v]
                prec[v] = u
    for i in xrange(n):
        if prec[i] is not None:
            ans.append((i,prec[i]))
    return ans
```

图 3: Prim 算法代码截图

说明，这里我的 Prim 算法并没有用最小堆来做（懒得做了……）。

来我们来看一下我写的 Kruskal 算法的部分。初始化时我用 python 中的 list 对象来表示连通的子集，于是每一个顶点 v，对应的初始集合是 $[v]$。edge 是边集，我用 python 内置的 sort 方法对 edge 按权重进行由低到高的排序。对于排序好的 edge 中的每一条边 e，我先得到了 e 的两个端点 u，v。我们先判断是否有 $sets[u] = sets[v]$（这里的正确性我们等会再说）。如果相等，表示 u 和 v 已经是连通的了，这条边不需要再加进来。如果不相等，那么我们把这条边加到最小生成树的边集中去（即 ans 集合）。然后进行对 u 和 v 所在的集合进行合并。合并分为四步。第一步，判断 $sets[u], sets[v]$ 的长度，如果后者大，交换 u，v。这一步的目的在于保证我们每一步都是较短集合加入较长集合。第二步，讲 $sets[v]$ 中的每一个元素 w 都加入到 $sets[u]$ 中，表示实际的集合合并过程。第三步，将 $sets[u]$ 的值赋给 $sets[v]$ 中的每一个元素 w，$sets[w] = sets[u]$。注意，这里是重点。我们不需要对 $sets[u]$ 中的所有其它元素对应的连通子集进行改变，原因是，python 中的有一个特性，它里面的任何变量都是一个引用。我们在进行对 $sets[u]$ 进行 append 操作时，已经对和 u 连通的顶点对应的 sets 进行了改变，通过这一点点特性，我们可以讲这部分的总时间复杂度变为 $V \log V = E \log V$。

实验分析

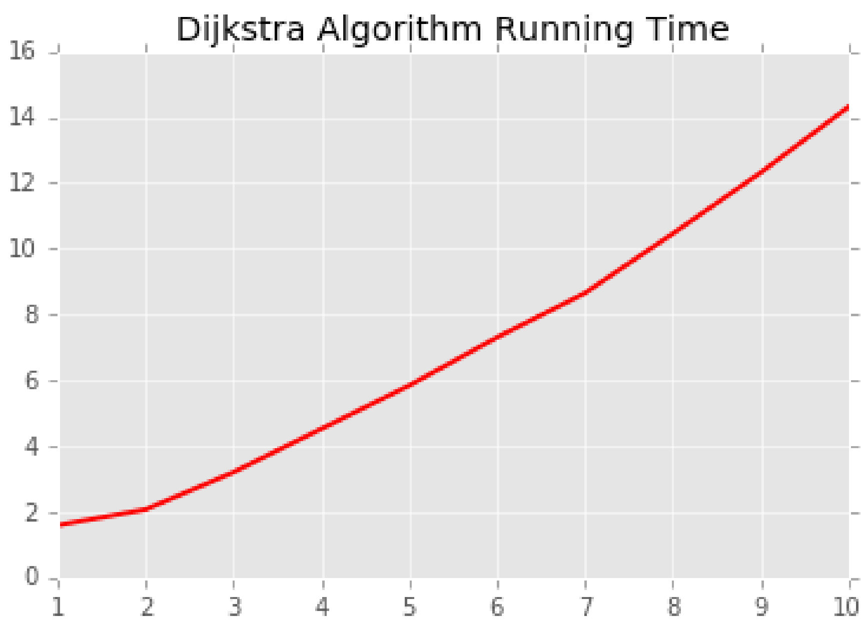

图 4: Dijkstra 算法运行时间图：$\log T - \log n$

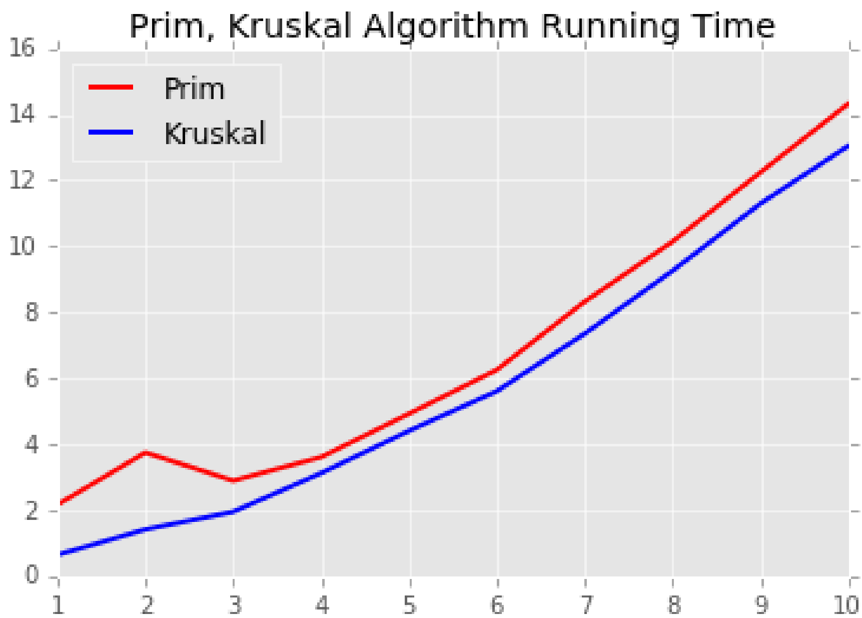

图 5: Prim，Kruskal 算法运行时间图

从实验结果可以看到，这几个算法的运行时间的对数和顶点数的对数基本上成线性关系，斜率差不多为 1.3 左右，这和理论上的时间复杂度还是比较契合的。

Kruskal 算法始终要比 Prim 算法要快一些，我猜测原因是我的 prim 算法没有用到堆来进行贪心选择，导致时间复杂度比 Kruskal 要高。

实验总结

这次实验做的还是蛮愉快的，对这几个经典算法有了更深得了解，对 python 的一些特性也多了一些了解。