HW04 贪心算法

22

01 背景知识

在求解最优解问题的过程中，依据某种贪心标准，从问题的初始状态出发，直接去求每一步的最优解，通过若干次的贪心选择，最终得出整个问题的最终解，这种求解方法就是贪心算法。贪心算法所做的选择可以依赖于以往所做过的选择，但绝不依赖于将来的选择，也不依赖于子问题的解。

贪心算法的基本要素：

- 贪心算法性质：所求问题的整体最优解可以通过一系列局部最优的选择（即贪心选择）来达到。
- 最优子结构性质：一个问题的最优解包含其子问题的最优解时。

  贪心算法适用的情型：
- 最优解可以通过局部最优解累积得到的问题： 如图的最小生成树、哈夫曼编码等。
- 作为其他算法的一部分： 在复杂问题中，贪心算法可以作为求解步骤的一部分，与其他算法如动态规划、回溯法结合使用。

02 算法实现

02-1 最大多位数问题

题目描述：

> 设有 n 个正整数，将他们连接成一排，组成一个最大的多位整数。
> 例如 :n=3 时， 3 个整数 13, 312, 343, 连成的最大整数为 :34331213，
> 又如 :n=4 时 , 4 个整数 7, 13, 4, 246 连接成的最大整数为 7424613

题目分析与算法设计：

- 看到题目很容易想到贪心法，但是也很容易搞错，容易想成把数字越大的放在最前面，最后组成的数字将是最大的，但是这样是错误的，比方两个数123和72,72123>12372，又或者12和121，应该组成12121而不可以是12112，又或者12,123，是12312。但是这并不意味着不可以用贪心算法，实际是把贪心标准找错了。
- 既然要保证每个数字连接之后最大，那么就保证连接后的数字最大，同时为了保证数据不溢出，我们可以使用字符串来存储数据，因为本题不涉及字符串内部操作，而且C++中可以使用‘+’将两个字符串进行拼接。

- 正确的贪心标准应该是：局部最优解：先把整数化成字符串，然后再比较a+b和b+a，如果a+b>b+a那么就把a排在b的前面，反之则把a排在b的后面。全局最优解：所有整数的连接顺序
- 然而表真的字符串比较存在缺陷，如：A=’321’，B=’32’，按照标准的字符串比较规则因为A>B，所以A+B > B+A，而实际上’32132’ <’32321’。所以我们需要自定义一种字符串比较规则：如果A+B>B+A,则认为A>B，且可以证明：如果A+B>=B+A,B+C>=C+B,则一定有A+C>=C+A;
  算法分三步：
- 先把n个数字转换成字符串存储在vector容器中；
- 再按照自定义的规则把n个字符串排序；
- 最后按照从小到大的顺序输出这些字符串
  复杂性分析：
  时间复杂度：
- **读取输入字符串**：读取n个字符串的复杂度为$O(n)$，但实际上，因为涉及到字符串的输入，这个步骤的时间复杂度还应该考虑字符串的平均长度m，所以更准确的时间复杂度是$O(nm)$.
- **排序操作**：排序算法是一个双重循环，外层循环运行$n-1$次，内层循环最多运行n次，所以基本的操作次数是$O(n^2)$。但是，排序中的比较操作涉及到字符串的连接和比较，假设平均字符串长度为m，那么每次比较的时间复杂度是$O(m)$。因此，排序步骤的总时间复杂度为$O(n^2m)$.
  综合上述步骤，整个程序的时间复杂度主要由排序步骤决定，即$O(n^2m)$.
  空间复杂度：
- 存储字符串的向量：向量num存储了n个字符串，如果我们假设字符串的平均长度为m，那么总的空间复杂度为$O(nm)$。
- 临时字符串：在排序过程中使用了临时字符串temp来交换字符串，但这不会影响总体的空间复杂度，因此可以忽略。
  因此，整个程序的空间复杂度为$O(nm)$。
  源代码：

```
#include<iostream>
#include<cstring>
#include<vector>
using namespace std;

int main(){
        vector<string> num;//使用vector容器存储
        int n = 0;
        cin >> n;
        for (int i = 0; i < n; i++) {
                string temp;
                cin >> temp;
```

```
                num.push_back(temp);
        }
        for (int i = 0; i < n - 1; i++) {
                for (int k = i + 1; k < n; ++k) {
                        if (num[i] + num[k] < num[k] + num[i]) {
                                string temp = num[i];//贪心算法求解局部最
优解
                                num[i] = num[k];
                                num[k] = temp;
                        }
                }
        }
        for (int i = 0; i != num.size(); i++) {
                cout<<num[i];
        }
}
```

02-2 最优服务次序问题

题目描述：

> 设有n个顾客同时等待一项服务，顾客i需要的服务时间为ti。应如何安排n个顾客的服务次序才能使平均等待时间达到最小?
> 平均等待时间是n个顾客等待直到完成服务的时间总和除以n。
> 对于给定的n个顾客需要的服务时间，编程计算最优服务次序。

题目分析与算法设计：
假设原问题为T（先假设只有一个服务点），而我们已经知道了某个最优服务系列，即最优解为A{t(1),t(2),...,t(n)}(其中t(i)为第i个用户需要的服务时间)，则每个用户等待时间为：
T(1)=t(1); T(2)=t(1)+t(2); .....T(n)=t(1)+t(2)+...+t(n);
那么总的等待时间，即最优值为：

$$\begin{aligned} T_A =& T(1)+T(2)+\cdots+T(n) \\ =& n*t(1)+(n-1)*t(2)+\ldots \\ & +(n+1-j)*t(i)+\cdots+2*t(n-1)+t(n); \end{aligned}$$

由于平均等待时间是n个顾客等待时间的总和除以n，故本题实际上就是求使顾客等待时间的总和最小的服务次序。
所以本问题采用贪心算法求解：
先证明问题拥有贪心选择的性质：即最优服务A中t(1)满足条件：$t(1)<=t(i)(2<i<n)$.
使用反证法来证明：假设t(1)不是最小的，不妨设$t(1)>t(i)(i>1)$.
设另一服务序列$B=(t(1),t(2),\ldots,t(i),\ldots,t(n))$

那么$TA-TB=n*[t(1)-t(i)]+(n+1-i)[t(i)-t(1)]=(1-i)*[t(i)-t(1)]>0$
即TA>TB,这与A是最优服务相矛盾，所以最优服务次序问题满足贪心选择性质。
问题的最优子结构性质：
在进行了贪心选择后，原问题T就变成了如何安排剩余的n-1个顾客的服务次序的问题T'，是原问题的子问题。若A是原问题T的最优解，则A'={t(2)，…t(i)…t(n))是服务次序问题子问题T'的最优解。
证明：假设A'不是子问题T'的最优解，其子问题的最优解为B'，则有$TB'<TA'$,而根据TA的定义知，$TA'+t(1)=TA$。因此$TB'+t(1)<TA'+t(1)=TA$,即存在一个比最优值TA更短的总等待时间，而这与TA为问题T的最优值相矛盾。因此，A'是子问题T'的最优值。
从以上贪心选择及最优子结构性质的证明，可知对最优服务次序问题用贪心算法可求得最优解。根据以上证明，最优服务次序问题可以用最短服务时间优先的贪心选择可以达到最优解.故只需对所有服务先按服务时间从小到大进行排序，然后按照排序结果依次进行服务即可。平均等待时间即为TA/n.
复杂性分析：
时间复杂度：

- 输入顾客的服务时间需要$O(n)$的时间复杂度，其中n是顾客的数量。
- 对服务时间进行排序，使用的是标准库中的排序算法，时间复杂度为$O(n\log n)$.
- 计算每个顾客完成服务的累计时间需要$O(n)$的时间，因为这一步通过简单的循环实现.
- 计算总等待时间同样需要$O(n)$的时间复杂度.
  综上所述，整个程序的主要时间复杂度由排序步骤决定，即$O(n\log n)$.
  空间复杂度：
- 程序中使用了两个数组t和s，它们的大小都是根据顾客数量n来确定的。因此，空间复杂度为$O(n)$.
  源代码：

```
#include<iostream>
#include<algorithm>
#include <iomanip>
using namespace std;

const int N = 100010;
int t[N],s[N];
int n;

int main(){
        cin >> n;// 输入顾客数量
    // 输入每个顾客需要的服务时间
        for (int i = 1; i ≤ n; ++i)
                cin >> t[i];
```

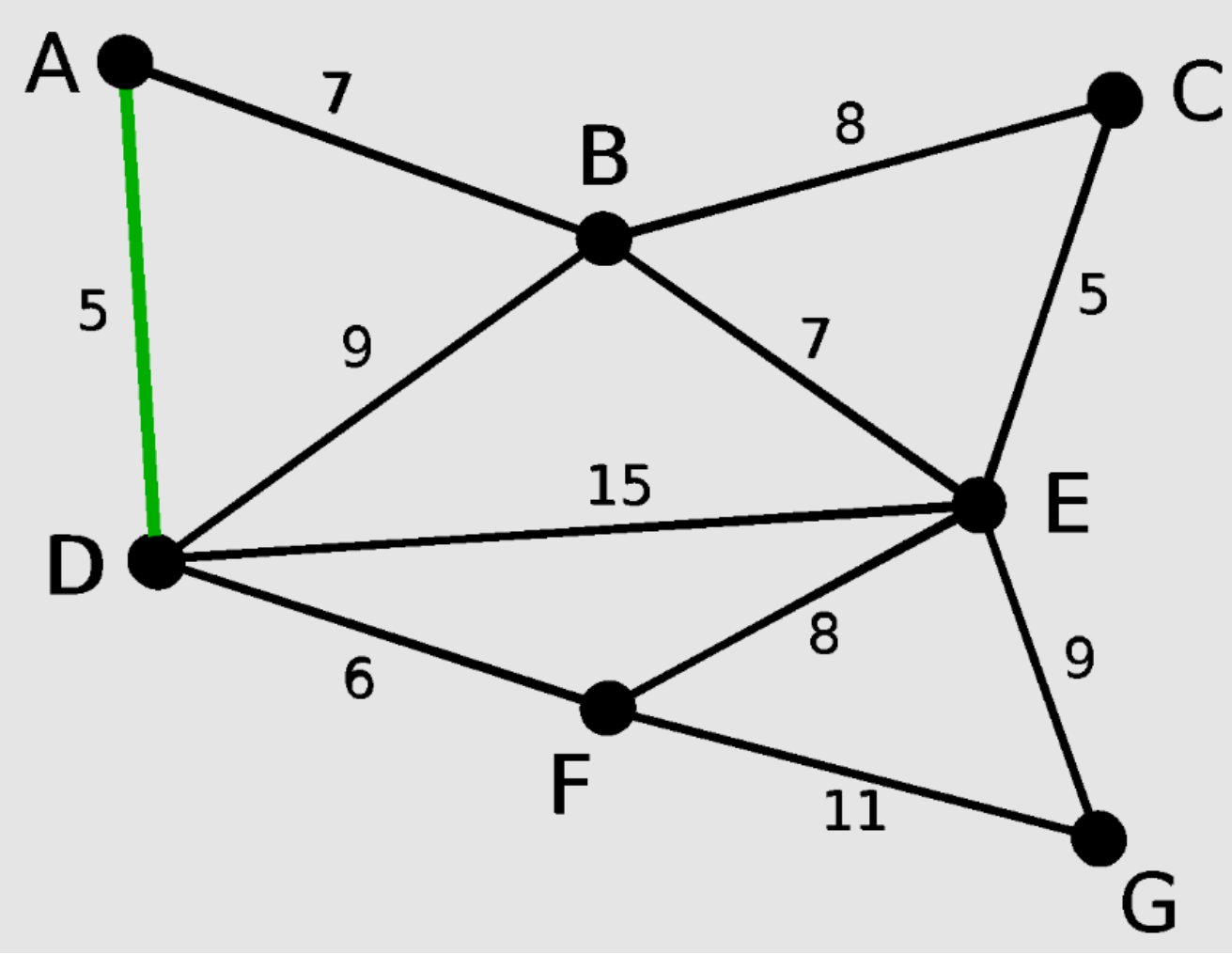
```
        sort(t + 1, t + n + 1);// 将顾客的服务时间按照升序排序
    // 计算每个顾客完成服务时的累计时间
        for (int i = 1; i ≤ n; ++i)
                s[i] = s[i - 1] + t[i];
        int res = 0;// 计算总等待时间
        for (int i = 1; i ≤ n; ++i)
                res += s[i];
    // 计算平均等待时间
        double ans = static_cast<double>(res) / n;
    // 输出结果，保留两位小数
        cout << fixed << setprecision(2) << ans << endl;
        return 0;
}
```

02-3 Kruskal算法实现MST

题目描述：

> 使用Kruskal算法实现最小生成树

题目分析与算法设计：
最小生成树的性质：其边数=顶点数-1，且树中一定不存在环。



算法思想：
Kruskal算法采用了边贪心策略，基本思路是在初始状态时隐藏掉图中所有的边，这样图中每个顶点都自成一个连通块，随后执行：

1. 对所有边按照边权大小从小到大进行排序
2. 按照边权从小到大一条一条地将边加入到当前最小生成树集合中。有两种情况：①如果加入的边在最小生成树中所连接的两个顶点不在同一个连通块中，也就是说加入这条边后在树中不会形成环，那就将边加入；②否则，则舍弃不加入这条边
3. 反复执行步骤二，指导最小生成树的边数等于总顶点数-1，或者是测试完所有的边时结束。结束时，如果最小生成树的边数小于总顶点数-1，则说明图不连通。
   Kruskal算法贪心性质的证明：
   使用归纳法，证明任何时候Kruskal算法选择的边集都被某棵MST所包含
   当算法刚开始时，显然成立，MST存在
   假设某时刻成立，当前边集为F，令T为这棵MST，考虑下一条加入的边e

- 如果e属于T，那么成立
- 否则，$T+e$一定存在某个环，考虑这个环上不属于F的另一条边f（一定只有一条）
  首先，f的权值一定不会比e小，不然f会在e之前被选取
  然后，f的权值一定不会比e大，不然$T+e-f$就是一颗=棵比T还优的生成树了
  所以，$T+e-f$包含了F，并且也是一颗最小生成树，归纳成立。
  MST的唯一性：
  对于 Kruskal 算法，只要计算为当前权值的边可以放几条，实际放了几条，如果这两个值不一样，那么就说明这几条边与之前的边产生了一个环（这个环中至少有两条当前权值的边，否则根据并查集，这条边是不能放的），即最小生成树不唯一。
  算法实现：
  算法利用贪心思路，虽然简单但是需要相应的数据结构来支持，具体来说，维护一个森林，查询两个结点是否在同一棵树中，连接两棵树。抽象一点地说，维护一堆集合，查询两个元素是否属于同一集合，合并两个集合。其中，查询两点是否连通和连接两点可以使用并查集维护。
  使用结构体来存储边集的顶点与权重：

```
struct Edge {
        int u, v; // 起始顶点u，终止顶点v
        int w; // 边的权重
};
```

并查集核心代码：

```
// 查找元素x所在集合的根节点的函数,使用路径压缩优化查找操作。
// father: 存储每个元素父节点的向量的引用。
```

```
// x: 要查找其所在集合根节点的元素。
// return: 包含x的集合的根节点。
int findFather(vector<int>& father, int x) {
        if(father[x] == x) return x;
        else
        return father[x] = findFather(father, father[x]); // 路径压缩
}
```

Kruskal核心代码：

```
// 执行Kruskal算法以找到图的最小生成树（MST）的函数。
// N: 图中顶点的数量。
// edges: 图中边的向量的引用。
// return: 如果存在MST，则返回MST的总权重；否则返回-1。
int kruskal(int N, vector<Edge>& edges) {
    vector<int> father(N + 1); // 并查集的父节点数组
    for (int i = 1; i ≤ N; ++i) // 初始化每个顶点为其自己的父节点
        father[i] = i;
    // 按权重非递减顺序对边进行排序
    sort(edges.begin(), edges.end(), [](const Edge& a, const Edge& b) {
        return a.w < b.w;
    });

    int ans = 0, numEdges = 0; // 初始化MST的总权重和已添加的边数
    for (const Edge& e : edges) {
        int fu = findFather(father, e.u); // 查找顶点u的根节点
        int fv = findFather(father, e.v); // 查找顶点v的根节点
        if (fu ≠ fv) { // 如果u和v在不同的集合中，将边添加到MST中
            father[fu] = fv; // 合并集合
            ans += e.w; // 将边的权重加到总权重中
            ++numEdges; // 已添加的边数加一
            if (numEdges == N - 1) break; // 如果MST已完成，退出循环
        }
    }
    return (numEdges == N - 1) ? ans : -1; // 如果MST存在，返回总权重；
}
```

复杂性分析：
时间复杂性主要取决于排序算法和并查集算法。

- 对边排序的操作时间复杂度为$O(M \log M)$，其中M是边的数量.
- 并查集操作$findFather$函数使用了路径压缩，使得平摊时间复杂度近乎常数，大约为$O(\alpha(N))$，其中$\alpha(N)$是$Ackermann$函数的逆函数。它是一个增长非常慢的函数，在所有实际情况下，它可以被认为几乎是常数。在最坏的情况下，每条边都会

引起两次$findFather$操作（每个顶点一次），使得这部分的时间复杂度为$O(2M\alpha(N)) = O(M\alpha(N))$.

- 遍历边时使用for循环，在最坏情况下位$O(M)$.
  所以Kruskal算法的整体时间复杂度可近似为$O(MlogM)$.
  空间复杂度主要由存储$edges$向量和$father$向量引起，给我们$O(M+N)$.
  源代码：

```
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;
struct Edge {
    int u, v;
    int w;
};
int findFather(vector<int>& father, int x) {
    if(father[x] == x)
        return x;
        else
        return father[x] = findFather(father, father[x]);
}
int kruskal(int N, vector<Edge>& edges) {
    vector<int> father(N + 1);
    for (int i = 1; i ≤ N; ++i)
        father[i] = i;

    sort(edges.begin(), edges.end(), [](const Edge& a, const Edge& b) {
        return a.w < b.w;
    });
    int ans = 0, numEdges = 0;
    for (const Edge& e : edges) {
        int fu = findFather(father, e.u);
        int fv = findFather(father, e.v);
        if (fu ≠ fv) {
            father[fu] = fv;
            ans += e.w;
            ++numEdges;
            if (numEdges == N - 1) break;
        }
    }
    return (numEdges == N - 1) ? ans : -1;
}
int main() {
    int N, M;
    cin >> N >> M;
```

```
    vector<Edge> edges(M);
    for (int i = 0; i < M; ++i) {
        cin >> edges[i].u >> edges[i].v >> edges[i].w;
    }
    int minWeight = kruskal(N, edges);
    if (minWeight != -1)
        cout << minWeight << endl;
    else
        cout << "impossible" << endl;
    return 0;
}
```

03 总结

- 本次作业涉及贪心算法的相关知识。贪心算法总是作出在当前看来最好的选择，即贪心算法并不从整体最优考虑，它所作出的选择只是在某种意义上的局部最优选择，不一定达到全局最优。 当一个问题动具有最优子结构性质时,可用动态规划算法求解，但有时贪心算法会更简单有效。
- 虽然贪心算法不是对所有问题都能从局部得到整体的最优解，但它的应用仍然很广。例如：哈夫曼编码、单源最短路径问题、最小生成树问题。特别是在一些情况下，最优解无法得到时，贪心算法却是能求得最优解的很好近似解。