《算法设计与分析》实验报告

学　号：1004161221　　姓　名：白云鹏

日　期：2019.01.08　　得　分：

一、实验内容：

TSP 问题。

二、所用算法的基本思想及复杂度分析：

1、蛮力法

1）基本思想

蛮力法的思想较为简单，即穷举所有的状态来找到最小花费的一个状态，这里的状态由城市标号的序列构成，在遍历的时候也只需要逐个计算相应的代价最后叠加即可得到每一个状态的总代价，最后找出所有状态中的最小的代价，所对应的状态即为最佳的答案。

2）复杂度分析

在外围循环中，需要穷举所有的状态，这里这个复杂度即为全排列所需的复杂度，为$\boldsymbol{\Theta}(\boldsymbol{n}!)$，但是在实际实现的过程中，固定第一个城市从 0 号城市出发，那么这里外层的实际复杂度为：

$$\boldsymbol{\Theta}\big((\boldsymbol{n}-\boldsymbol{1})!\big)$$

而在内层循环中，对于每一个状态而言，都需要遍历状态数组得到结果，由于固定了第一号元素，那这里的复杂度为：

$$\boldsymbol{\Theta}(\boldsymbol{n}-\boldsymbol{1})$$

最后最终实际的复杂度为：

$$\boldsymbol{\Theta}((\boldsymbol{n}-\boldsymbol{1})^{\boldsymbol{2}}*(\boldsymbol{n}-\boldsymbol{2})!)$$

2、动态规划法

1）基本思想

在所实现的动态规划法中，使用了状态压缩的思想，将每一个状态利用数的二进制进行表示。那么当前路径已经经过的点就可以用二进

制每一位是否为 1 来表示，因为在每一个状态中，这次要遍历的节点只与上一次到达的节点以及整体的标记有关，那么第一维只需要为其维护当前状态遍历过的节点，这里用数字的二进制表示，因此最大只需要$(\mathbf{1} \ll \boldsymbol{n})$的大小，而第二维度则维护其上一次到达的节点，大小为$\boldsymbol{n}$。在遍历的过程中，如果这个节点已经在当前状态中被标记，即$(\mathbf{1} \ll \boldsymbol{k})\&\boldsymbol{status} != \mathbf{0}$，那么跳过这个节点，否则就需要根据当前状态来更新新的状态，状态的转移为：

$$\boldsymbol{dp}[(\mathbf{1} \ll \boldsymbol{k}) + \boldsymbol{status}][\boldsymbol{j}] = \boldsymbol{min}(\boldsymbol{dp}[\boldsymbol{status}][\boldsymbol{i}] + \boldsymbol{m}[\boldsymbol{i}][\boldsymbol{k}], \boldsymbol{dp}[(\mathbf{1} \ll \boldsymbol{k}) + \boldsymbol{status}][\boldsymbol{j}]$$

2）复杂度分析

在代码中既可以看出，这里的复杂度为：

$$\boldsymbol{\Theta}(\mathbf{2}^{\boldsymbol{n}} * \boldsymbol{n}^{\mathbf{2}})$$

3、分支限界法

1）基本思想

分支界限法是按照宽度对解空间树进行遍历，这即是每一次在进行搜索的时候都是用 BFS 将最优的节点取出然后在其基础上进行节点的更新。因此，一个合适的界限函数就是一个十分重要的判断依据。在实现的过程中，使用贪心法来寻找解的上界。对于下界，考虑每一个可行解，显然每一个城市有且只有一次进入以及一次离开，一条路径构成了一个环，因此在寻找下界的时候，从最近的城市进入再去最近的城市，再将所得到的值除以 2 向上取整作为下界。对于部分解而言，其下界就为已经过的路径长度的二倍再加上从起点到最近的未遍历的城市的长度以及从终点到最近未遍历的城市的长度，再将这个长度除以 2 向上取整。

2）复杂度分析

在 BFS 的过程中，最坏的情况就是将所有的节点都进行遍历，这里的复杂度显然是：

$$\boldsymbol{\Theta}(\mathbf{n}!)$$

4、回溯法

1）基本思想

按照一定的顺序对所有的情况进行穷举。逐一判断每一个情形是否是

合法的，在合法的前提下再在所有的情况中寻找所有路程的最小值。

在实现中，使用了 DFS 作为遍历的顺序。首先在初始化的函数中主要是清空了标记数组以及输入边权进行图的构建。接下来在 DFS 函数中就是进行穷举的过程。首先 DFS 的三个参数分别表示上一次到达的城市编号，已经经过的路程长度以及已经经过的城市个数。进入 DFS 流程之后，首先是检查是否是一个解，如果是一个解，进行更新答案之后返回。如果不是的话就需要穷举下一次 DFS 的城市标号，将这个点标记以后再进行搜索，最后取消标记。

2）复杂度分析

在整个的遍历过程中，可以看出需要穷举所有的路径，这里所有的答案即为将标号进行全排列，这个复杂度为：

$$\Theta(n!)$$

其中 n 代表城市的个数。

三、源程序及注释：

1、蛮力法

```
#include <bits/stdc++.h>

using namespace std;


const int maxn = 12;//最多城市数

vector<int> vv;//枚举的状态顺序数组

int mm[maxn][maxn];

int n;//当前样例城市数


void init() {//初始化函数

    vv.clear();

    for (int i = 0; i < n; ++i) {

        for (int j = 0; j < n; ++j) {

            if (i == j) {

                mm[i][j] = 0;

                continue;
```

```
            }
            cin >> mm[i][j];
        }
    }
}

int cal(const vector<int> &t) {//计算当前状态花费的函数
    int pre = t[0];//记录前一次出发的城市
    int ans = 0;//此次状态的花费
    for (int i = 1; i < t.size(); ++i) {
        ans += mm[pre][t[i]];
        pre = t[i];
    }
    ans += mm[pre][t[0]];//最后需要回到出发城市
    return ans;
}

int solv() {
    for (int i = 0; i < n; ++i) {
        vv.push_back(i);
    }
    int min1 = INT_MAX;
    do {
        min1 = min(cal(vv), min1);
    } while (next_permutation(vv.begin() + 1, vv.end()));//对所有的状态进行全排列
    return min1;
}
int main() {

    while (cin >> n) {
```

```
        init();
        cout << solv() << endl;
    }
}
```

2、动态规划法

```
#include <bits/stdc++.h>
using namespace std;
const int inf = 0x3f3f3f;//无穷大
int n;
int dp[1 << 17][17];//dp
int mp[17][17];
int main() {
    while (cin >> n) {
        int ans = inf;
        //输入边权进行建图
        for (int i = 1; i <= n; i++) {
            for (int j = 1; j <= n; j++) {
                if (i == j) {
                    continue;
                }
                cin >> mp[i][j];
            }
        }
        //将数组置为无穷
        memset(dp, inf, sizeof(dp));
        //初始为0
        dp[1][1] = 0;
        //最大的状态数为(1<<n)
        int maxst = (1LL << n);
       //穷举所有已经经过的城市
        for (int i = 1; i < maxst; ++i) {
            //这一次所访问的城市
            for (int j = 1; j <= n; ++j) {
                int q = 1 << (j - 1);
```

```
            if ((i & q) == 0) {
                //如果当前城市没有来过
                for (int k = 1; k <= n; ++k) {
                    //以该城市为节点进行更新
                    dp[i + q][j] = min(dp[i + q][j], dp[i][k] + mp[k][j]);
                }
            }
        }
    }
    //最后需要再回到出发节点
    for (int i = 2; i <= n; i++) ans = min(ans, dp[(1<<n)-1][i] + mp[i][1]);
    cout << ans << endl;
  }
}
```

3、分支限界法

```
#include <algorithm>
#include <cstring>
#include <cmath>
#include <cstdio>
#include <iostream>
#include <queue>
using namespace std;
const int INF = 10000000;
int low, up, n, used[20], graph[20][20];
struct node {
    bool vis[20];
    int st;
    int ed;
    int k;
    int sumv;
    int lb;
    bool operator<(const node &p) const {
        return lb > p.lb;
    }
};
priority_queue<node> q;
void getup() {//贪心法寻找上界
    used[1] = 1;
    int kk = 1;
```

```
    int len = 0;
    int minlen;
    int pre = 1;
    while (kk != n) {
        minlen = INF;
        for (int i = 1; i <= n; ++i) {
            if (used[i] == 0 && minlen > graph[pre][i]) {
                minlen = graph[pre][i];
                pre = i;
            }
        }
        used[pre] = 1;
        ++kk;
        len += minlen;
    }
    len += graph[pre][1];
    up = len;
}
int getlb(node p) {//寻找下界操作
    int ret = p.sumv * 2;
    int min1 = INF, min2 = INF;
    for (int i = 1; i <= n; i++) {
        if (p.vis[i] == 0 && min1 > graph[p.st][i]) {
            min1 = graph[p.st][i];
        }
    }
    ret += min1;
    for (int i = 1; i <= n; i++) {
        if (p.vis[i] == 0 && min2 > graph[p.ed][i]) {
            min2 = graph[p.ed][i];
        }
    }
    ret += min2;
    for (int i = 1; i <= n; i++) {
        if (p.vis[i] == 0) {
            min1 = min2 = INF;
            for (int j = 1; j <= n; j++) {
                if (min1 > graph[i][j]) min1 = graph[i][j];//从当前城市出发到最近的城市
            }
            for (int j = 1; j <= n; j++) {
                if (min2 > graph[j][i]) min2 = graph[j][i];//从最近的城市到当前城市
            }
            ret += min1 + min2;
        }
```

```
    }
    return (ret + 1) /2; //除 2 后向上取整
}

void getlow() {//全局第一个节点的下界
    low = 0;
    for (int i = 1; i <= n; i++) {
        int temp[20];
        for (int j = 1; j <= n; j++) {
            temp[j] = graph[i][j];
        }
        sort(temp + 1, temp + 1 + n);
        low = low + temp[1] + temp[2]; //取最近的两个城市的距离
    }
    low = low / 2;
}

int solve() {
    getup();
    getlow();
    node start;
    start.st = 1;
    start.ed = 1;
    start.k = 1;
    for (int i = 1; i <= n; i++) start.vis[i] = 0;
    start.vis[1] = 1;
    start.sumv = 0;
    start.lb = low;
    int ret = INF;
    q.push(start);
    node next, temp;
    while (!q.empty()) {
        temp = q.top();
        q.pop();
        if (temp.k == n - 1) //如果是最后一个城市
        {
            int pos = 0;
            for (int i = 1; i <= n; i++) {
                if (temp.vis[i] == 0) {
                    pos = i;
                    break;
                }
            }
            if (pos == 0) break;//如果这个城市非法
```

```
            //取当前节点的长度
            int ans = temp.sumv + graph[pos][temp.st] + graph[temp.ed][pos];
            node judge = q.top();
            if (ans <= judge.lb) {//判断是否是最小
                ret = min(ans, ret);
                break;
            }
            else {//更新上界
                up = min(up, ans);
                ret = min(ret, ans);
                continue;
            }
        }
        for (int i = 1; i <= n; i++) {//否则就取下一个顶点继续进行搜索
            if (temp.vis[i] == 0) {
                next.st = temp.st;
                next.sumv = temp.sumv + graph[temp.ed][i];
                next.ed = i;
                next.k = temp.k + 1;
                for (int j = 1; j <= n; j++) next.vis[j] = temp.vis[j];
                next.vis[i] = 1;
                next.lb = getlb(next);
                if (next.lb >= up) continue;
                q.push(next);
            }
        }
    }
    return ret;
}

int main() {
    ios::sync_with_stdio(false);
    while (cin >> n) {
        int x[20], y[20];
        memset(graph, 0, sizeof graph);
        memset(used, 0, sizeof used);
        for (int i = 1; i <= n; i++) {
            for (int j = 1; j <= n; j++) {
                if (i == j)
                    graph[i][j] = INF;
                else
                    cin >> graph[i][j];
            }
        }
```

```
        cout << solve() << endl;
    }
}
```

4.回溯法

```
#include <bits/stdc++.h>

using namespace std;

int m[20][20];

int mark[20];//dfs 标记数组，对 dfs 过程中经过的节点进行标记

int n;//城市数目

int ans = INT_MAX;//最终结果

void init() {

    ans = INT_MAX;//初始化为最大

    memset(mark, 0, sizeof mark);//清空标记数组

    //读入

    for (int i = 0; i < n; ++i) {

        for (int j = 0; j < n; ++j) {

            if (i == j) continue;

            int tem;

            cin >> tem;

            m[i][j] = tem;

        }

    }

}

void dfs(int i, int len, int sz = 0) {//第一个参数表示当前城市标号，第二个参数表示当前路径的长度，第三个参数表示已经经过的城市个数

    if (sz == n && i == 0) {

        ans = min(ans, len);//更新最小值

        return;

    }

    if(sz > ans)return;//剪枝操作，如果当前路径的长度已经比答案大，直接返回

    for (int j = 0; j < n; ++j) {
```

```
        if (j == 0 && sz != n - 1) continue;//剪枝操作，如果当前经过的城市个数不为 n-1 但是已经回到了起点，剪掉
        if (j == i) continue;
        if (mark[j] == 0) {
            mark[j] = 1;//标记已经来过
            dfs(j, len + m[i][j], sz + 1);//向下 dfs
            mark[j] = 0;//取消标记
        }
    }
}
void solv() {
    dfs(0, 0);
    cout << ans << endl;
}
int main() {
    ios::sync_with_stdio(0);
    while (cin >> n) {
        init();
        solv();
    }
}
```

四、运行输出结果：

```
5
3 1 5 8
3 6 7 9
1 6 4 2
5 7 4 3
8 9 2 3
16
10
       42  160   34  136  134   94   78   18  196
   42      166   66  106   87   11  122  195   32
  160  166       4   98  198    3  154   75  121
   34   66    4      187  112   52   94   36  144
  136  106   98  187       12   64   45   46   48
  134   87  198  112   12       154  109  196  131
   94   11    3   52   64  154       11   79   80
   78  122  154   94   45  109   11       13   86
   18  195   75   36   46  196   79   13       162
  196   32  121  144   48  131   80   86  162
284
```

图 1 四种方法的运行结果

四种方法的运行结果均如上所示，不再赘述。

五、调试和运行程序过程中产生的问题、采取的措施及获得的相关经验教训：

1、在整个问题的测试中，首先需要注意的是每一个城市不能再经过自己这个城市，所以在建图的时候，可以把边权置为正无穷大表示不能从当前节点回到这个节点。

2、在蛮力法的调试中，需要注意的是每次在 dfs 前要将节点标记起来，在搜索结束后需要清楚标记，否则就不能遍历完所有的状态。

3、在动态规划法的调试中，由于已经有了状态转移方程，所以在进行设计的时候，主要是初始状态的设置比较困难。在完成了递推之后需要注意的是还需要再加上回到初始节点的路程。

4、在分支界限法的调试中，如果搜索到第一个合法解的时候就结束流程，可能会陷入局部最优而非全局最优。

5. 在回溯法中则需要注意在每一次搜索结束后，需要讲此次标记的城市取消以避免影响下次的搜索。没有取消，就无法搜索到全部的解并且很快就进行返回，答案显然是不正确的。

六、算法与代码的对应与解释（抽讲）

- 蛮力法：

在蛮力法中主要是用了下面这个函数来穷举所有的遍历顺序。

```
next_permutation(vv.begin(),vv.end());
```

这个函数是将所传递的容器对其进行全排列，由此就省去回溯法枚举其全排列的过程，简化程序。

- 动态规划法

动态规划法的关键部分主要是状态转移部分。

```
for (int i = 1; i < maxst; ++i) { //首先是用二进制方式枚举所有的状态
          for (int j = 1; j <= n; ++j) {//枚举所有的城市
              int q = 1 << (j - 1);
              if ((i & q) == 0) {
                  //如果当前城市没有来过
```

```
                for (int k = 1; k <= n; ++k) {
                    //以该城市为节点进行更新
                    dp[i + q][j] = min(dp[i + q][j], dp[i][k] + mp[k][j]);
                }
            }
        }
    }
```

- 分支界限法

分支界限法的关键部分则是上下界的寻找：

在上界的寻找过程中，实际上就是不断寻找离当前城市最近且没有被访问过的城市：

```
void getup() {//贪心法寻找上界
    used[1] = 1;
    int kk = 1;
    int len = 0;
    int minlen;
    int pre = 1;
    while (kk != n) {
        minlen = INF;
        for (int i = 1; i <= n; ++i) {
            if (used[i] == 0 && minlen > graph[pre][i]) {//如果这个城市没有被访问过并且比较近
            //更新最小值并记录这个城市的标号
                minlen = graph[pre][i];
                pre = i;
            }
        }
        used[pre] = 1;
        ++kk;
        len += minlen;
    }
    //最后需要返回起始城市
    len += graph[pre][1];
    up = len;
}
```

寻找部分解的下界则是将一个解处理成一个环，每一个城市都有且只有一次进入和离开：

```
int getlb(node p) {//寻找下界操作
    int ret = p.sumv * 2;
    int min1 = INF, min2 = INF;
    for (int i = 1; i <= n; i++) {
        if (p.vis[i] == 0 && min1 > graph[p.st][i]) {
            min1 = graph[p.st][i];
        }
```

```
    }
    ret += min1;
    for (int i = 1; i <= n; i++) {
        if (p.vis[i] == 0 && min2 > graph[p.ed][i]) {
            min2 = graph[p.ed][i];
        }
    }
    ret += min2;
    for (int i = 1; i <= n; i++) {
        if (p.vis[i] == 0) {
            min1 = min2 = INF;
            for (int j = 1; j <= n; j++) {
                if (min1 > graph[i][j]) min1 = graph[i][j];//从当前城市出发到最近的城市
            }
            for (int j = 1; j <= n; j++) {
                if (min2 > graph[j][i]) min2 = graph[j][i];//从最近的城市到当前城市
            }
            ret += min1 + min2;
        }
    }
    return (ret + 1) /2; //除 2 后向上取整
}
```

- 回溯法

  回溯法的主要部分即为 DFS 的过程：

```
void dfs(int i, int len, int sz = 0) {

    if (sz == n && i == 0) {//如果当前已经遍历了所有的城市并且回到了出发的城市

        ans = min(ans, len);//更新最小值

        return;

    }

    if(sz > ans)return;//剪枝操作，如果当前路径的长度已经比当前答案大，直接返回

    for (int j = 0; j < n; ++j) {

        if (j == 0 && sz != n - 1) continue;//剪枝操作，如果当前经过的城市个数不为 n-1 但是已经回到了起点，剪掉

        if (j == i) continue;

        if (mark[j] == 0) {

            mark[j] = 1;//标记已经来过

            dfs(j, len + m[i][j], sz + 1);//向下 dfs
```

```
            mark[j] = 0;//取消标记
        }
    }
}
```

在回溯的过程中有两次剪枝操作，DFS 的复杂度很高，在早期进行剪枝就能减少很大的消耗。DFS 的过程也比较清晰直观，首先就是返回操作，如果当前是一条合法路径，更新最小值以后返回，否则选择一个没有经过的城市，将其标记向下递归，在当前 DFS 完成之后需要取消标记防止影响下一次 DFS 过程。