中国地质大学（北京）

算法设计与分析

期末报告

学　　院：信息工程学院

专　　业：计算机科学与技术

班　　级：10041912

学　　号：1004191211

姓　　名：郎文鹏

联系方式：15902253523

邮　　箱：2300546456@qq.com

指导老师：季晓慧

日　　期：2021 年 12 月 5 日

目录

实验目的

算法设计与分析是计算机科学技术中处于核心地位的一门专业基础课，越来越受到重视，无论是计算科学还是科学实践，算法都在其中扮演者重要的角色。算法被公认为是计算机科学的基石，对于计算机专业学生而言学会读懂算法、设计算法应该是一项最基本的要求，本课程最后要求同学利用课上所学的算法设计分析方法与不同的经典算法设计求解 TSP 问题 0-1 背包问题与数独问题并进行分析。

实验内容

TSP问题

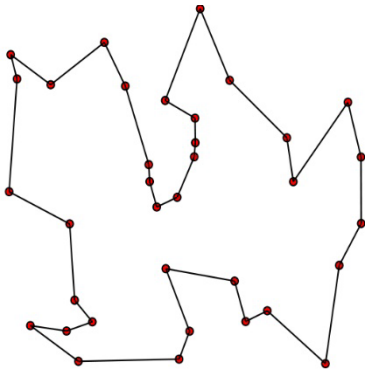

图 1：TSP示意图

旅行商问题，即TSP问题又译为旅行推销员问题、货郎担问题，是数学领域中著名问题之一。假设有一个旅行商人要拜访n个城市，他必须选择所要走的路径，路径的限制是每个城市只能拜访一次，而且最后要回到原来出发的城市。路径的选择目标是要求得的路径路程为所有路径之中的最小值。

蛮力法求解

算法想法

蛮力法求解 TSP 问题的基本思想是找出所有可能的旅行路线，即依次考察图中所有顶点的全排列，从中选取路径长度最短的哈密顿回路（也称为简单回路）。

哈密顿回路是著名的爱尔兰数学家哈密顿提出的周游世界问题。

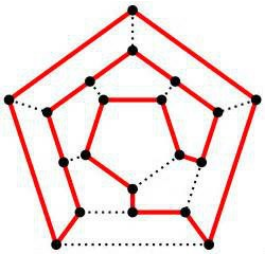

图 2：哈密顿回路示意图

假设一个正十二面体的 20 个顶点代表 20 个城市，哈密顿回路问题要求从一个城市出发，经过每个城市恰好一次，然后回到出发城市。模型抽象定义即：对于给定的无向图$G=(V,E)$，依次考察图中所有顶点的全排列满足以下两个条件的全排列构成的是哈密顿回路：

1）相邻顶点之间存在边，即$(v_{ij},v_{ij+1})\in E(1\le j\le n-1)$；

2）最后一个顶点和第一个顶点之间存在边，即$(v_{in},v_{i1})\in E$。

算法设计

蛮力法求解TSP问题与求解哈密顿回路问题类似，不同的仅是将回路经过的边上的权值相加得到相应的路径长度。伪代码如下：

算法：蛮力法求解TSP问题
输入：带权有向图$G=(V,E)$
输出：输出路径权值和最小的哈密顿回路
1. 对顶点集合$\{1,2,\ldots,n\}$的每一个排列$v_{i1},v_{i2},v_{i3}\ldots v_{in}$执行下述操作：
1.1 循环变量j从$1\sim n-1$重复执行下述操作

1.1.1 如果顶点v_{ij}和v_{ij+1}之间不存在边，则看做加上inf；

1.1.2 否则就加上边权；

1.2 用本次哈密顿回路的边权和更新答案；

2. 输出求解。

算法分析

蛮力法求解TSP问题必须依次考察顶点集和的所有全排列，从中找出路径长度最短的简单回路，因此时间下界是$O(n!)$。

源代码

```
#include <bits/stdc++.h>
#pragma GCC optimize(2)
#pragma G++ optimize(2)
#define endl "\n"
using namespace std;
typedef long long ll;
typedef pair<int, int> pii;
const int maxn = 105;

int dis[maxn][maxn];
void solve() {
    int n, S;
    cin >> n >> S;
    memset(dis, 0x3f, sizeof dis);
    for (int i = 1; i <= n; ++i)
        for (int j = 1; j <= n; ++j)
            cin >> dis[i][j];
    int a[n + 1] = {};
    for (int i = 1; i <= n; ++i) a[i] = i;
    int ans = 0x3f3f3f3f;
    do {
        if (a[1] != S) break;
        int temp = 0;
        for (int i = 2; i <= n; ++i)
            temp += dis[a[i - 1]][a[i]];
        temp += dis[a[n]][a[1]];
        ans = min(ans, temp);
```

```
        // for (auto &e : a) cout << e << ' ';
        // cout << temp << endl;
    } while (next_permutation(a + 1, a + 1 + n));
    cout << ans << endl;
}

signed main() {
    ios::sync_with_stdio(false), cin.tie(0);
#ifdef LOCAL
    freopen("IO\\in.txt", "r", stdin);
    freopen("IO\\out.txt", "w", stdout);
    clock_t start, end;
    start = clock();
#endif
    solve();
#ifdef LOCAL
    end = clock();
    cout << endl
         << "Runtime: " << fixed << setprecision(3) << (double)(end - sta
rt) << "ms\n";
#endif
    return 0;
}
```

算法结果

表 1：TSP 问题蛮力法求解结果

城市数量	计算结果	运行时间
5	1033	0ms
10	981	18ms
20	--	>30years
50	--	$> 4^{56} years$

（输入输出文件见附录）

由于算法的时间复杂度为阶乘级别，随着n的增大所需的计算量显著提升，虽然蛮力法的想法非常简单且编程易于实现，但其糟糕的时间复杂度有待优化。

动态规划法求解

算法想法

想要使用动态规划算法首先需要证明TSP问题满足最优性原理。设$s, s_1, s_2, \ldots, s_p$，s是从s出发的一条路径长度最短的简单回路，假设从s到下一个城市s_1已经求出，则问题转化为求从s_1到s的最短路径，显然$s_1, s_2, \ldots, s_p, s$一定构成一条从s_1到s的最短路径，如若不然，设$s_1, r_1, r_2, \ldots, r_q, s$是一条从$s_1$到$s$的最短路径且经过$n-1$个不同城市，则$s, s_1, r_1, r_2, \ldots, r_q, s$将是一条从$s$出发的路径长度最短的简单回路且比$s_1, r_1, r_2, \ldots, r_q, s$要短，从而导致矛盾。所以$TSP$问题满足最优性问题得证。

如何定义子问题？对于图$G=(V,E)$，假设从顶点i出发，令$V'=V-i$，则$d(i,V')$表示从顶点i出发经过V'中各个顶点一次且仅一次，最后回到出发点i的最短路径长度，显然初始子问题是$d(k,\{\})$，即从顶点i出发只经过顶点k回到顶点i。现在考虑原问题的一部分，$d(k,V'-\{k\})$表示从顶点k出发经过$V-\{k\}$中各个顶点一次且仅一次，最后回到出发点i的最短路径长度，则：

$$\begin{cases} d(k,\{\}) = c_{ki} \\ d(i,V') = min\{c_{ik} + d(k,V'-\{k\}))\} \end{cases}$$

算法设计

假设n个顶点分别用$1\sim n$数字编号，顶点之间的代价存放在数组$c[][]$中，下面考虑从顶点 1 出发求解TSP问题的填表形式。首先按照个数为$2,3,\ldots,n$的顺序生成$2\sim n$的顺序生成$2\sim n$个元素的子集存放在数组$V[2^{n-1}]$中。设数组$dp[n][2^{n-1}]$存放迭代结果，其中$dp[i][j]$表示从顶点$i+1$经过子集$V[j+1]$中的顶点中的顶点且仅一次，最后回到出发点 0 的最短路径长度，伪代码如下：

算法：动态规划法求解TSP问题
输入：图的代价矩阵$c[n][n]$
输出：从顶点 1 出发经过所有顶点一次且仅一次再回到顶点 1 的最短路径长度

1. 初始化边界，从顶点 1 未经过所有顶点的代价为 0；
2. 依次枚举每个状态下经过的每个顶点
   2.1 如果当前顶点在枚举状态中已经访问过则继续下一次迭代；
   2.2 否则利用状态方程进行转移；
3. 从最终所有数组中挑选再回到起点所需要的权值和最小作为答案输出。

算法分析

需要对顶点集合的每一个子集进行操作，因此时间复杂度为$O(2^n)$，和蛮力法相比动态规划法求解TSP问题把原来的时间复杂度为阶乘的排列问题转化为了组合问题从而降低了算法的时间复杂性，但是它仍然需要很大的计算量。

源代码

```
#include <bits/stdc++.h>
#pragma GCC optimize(2)
#pragma G++ optimize(2)
#define endl "\n"
using namespace std;
typedef long long ll;
const int maxn = 20;

int c[maxn][maxn];
int dp[1 << maxn][maxn];

void solve() {
    int n, S;
    cin >> n >> S;
    memset(dp, 0x3f, sizeof dp);
    //存边
    for (int i = 0; i < n; ++i)
        for (int j = 0; j < n; ++j)
            cin >> c[i][j];
    //初始化边界
    --S;
    dp[1 << S][S] = 0;
    for (int i = 0; i < (1 << n); ++i) {  //枚举状态
```

```
        for (int j = 0; j < n; ++j) {
            if (i & (1 << j)) {           //枚举出发点
                for (int k = 0; k < n; ++k) {  //枚举下一个点
                    if (i & (1 << k))
                        continue;
                    else
                        dp[i | (1 << k)][k] = min(dp[i | (1 << k)][k], dp
[i][j] + c[j][k]);
                }
            }
        }
    }
    int ans = 0x3f3f3f3f;  //答案
    for (int i = 0; i < n; ++i)
        ans = min(ans, dp[(1 << n) - 1][i] + c[i][S]);
    cout << ans << endl;
}

signed main() {
    ios::sync_with_stdio(false), cin.tie(0);
#ifdef LOCAL
    freopen("IO\\in.txt", "r", stdin);
    freopen("IO\\out.txt", "w", stdout);
    clock_t start, end;
    start = clock();
#endif
    solve();
#ifdef LOCAL
    end = clock();
    cout << endl
         << "Runtime: " << (double)(end - start) / CLOCKS_PER_SEC << "s\n
";
#endif
    return 0;
}
```

算法结果

表 2：TSP 问题动态规划法求解结果

城市数量	计算结果	运行时间

5	1033	37ms
10	981	44ms
20	800	852ms
50	--	--

（输入输出文件见附录）

存储集合的一个简单方法是借助二进制进行状态压缩，受限于自身电脑环境问题当城市数量达到 50 个时无法进行存储，由此可见TSP问题虽然优化了时间上的复杂性但是同样对空间上的要求进一步提高。

贪心法求解

算法想法

TSP问题采用合理的贪心策略对于某些情况同样可以获得正解且时间复杂度大幅度降低。常用的有两种普适的想法：

【想法 1】采用最近邻点策略：从任意城市出发，每次在没有到过的城市中选择最近的一个，直到经过了所有的城市，最后回到出发城市。

【想法 2】采用最短链接策略：每次在整个图的范围内选择最短边加入解集合中，但是要保证加入解集合中的边最终形成一个哈密顿回路。因此，当从剩余边集E'中选择一条边(u,v)加入解集合S中，应满足以下条件：

① 边(u,v)是边集E'中代价最小的边；

② 边(u,v)加入解集合S后，S中不产生回路；

③ 边(u,v)加入解集合S后，S中不产生分歧。

算法设计

由于我的测试数据为随机产生，对于想法 2 不好要求的图不好生成，因此这里以想法 1 的实现作为代表，源代码如下：

算法：贪心法最近邻点策略求解TSP问题
输入：有向带权图$G=(V,E)$，顶点w

输出：回路长度TSP最小权值和

1. 初始化：标记数组vis为空，记录答案$ans = 0$；
2. $u = w; vis[w] = 1;$
3. 循环直到标记数组vis为空，即包含了$n-1$条边

   3.1 查找与顶点u临接的最小代价边(u, v)并且v未被标记；

   3.2 $vis[v] = 1; ans += G[u][v];$
4. 输出答案。

算法分析

上述贪心策略所得到的结果容易陷入局部最优从而未必得到的结果是正解，只能保证在一定程度上贴近最优解，至于近似到何种程度难以得到保证。

源代码

```
#include <bits/stdc++.h>
using namespace std;
const int inf = 0x3f3f3f3f;
const int maxn = 55;
int G[maxn][maxn];
bool vis[maxn];

void solve() {
    int n, S;
    cin >> n >> S;
    for (int i = 1; i <= n; ++i)
        for (int j = 1; j <= n; ++j)
            cin >> G[i][j];
    vis[1] = 1;  //起点从 1 出发
    int ans = 0, now = 1, node, minn;
    for (int i = 1; i < n; ++i) {
        node = now, minn = inf;
        for (int j = 1; j <= n; ++j)
            if (!vis[j] && G[now][j] < minn) minn = G[now][j], node = j;
        vis[node] = 1, now = node, ans += minn;
        // cout << '(' << minn << ")->" << node << "-";
```

```
    }
    // cout << '(' << G[now][1] << ")->" << 1 << endl;
    ans += G[node][1];
    cout << ans << endl;
}

signed main() {
    ios::sync_with_stdio(false), cin.tie(0);
#ifdef LOCAL
    freopen("IO\\in.txt", "r", stdin);
    freopen("IO\\out.txt", "w", stdout);
    clock_t start, end;
    start = clock();
#endif
    solve();
#ifdef LOCAL
    end = clock();
    cout << endl
         << "Runtime: " << (double)(end - start) << "ms\n";
#endif
    return 0;
}
```

算法结果

表 3：TSP 问题贪心法法求解结果

城市数量	计算结果	运行时间
5	1033	0ms
10	1246	0ms
20	1761	0ms
50	1691	2ms

（输入输出文件见附录）

从测试结果来看，贪心策略在某些合适的情况下也有可能获得最优解，但是在关系较为复杂的图影响下逐渐偏离正解。这个思路提供了我们一个新的视角看待问题，针对特殊问题可以采用一些特殊的解法虽不能一步到位但是可以帮助我们能够有效的确定问题解的大致方向，在此基础上进行改良优化也是一个值得研

究的方向。

回溯法求解

算法思想

假定图$G=(V,E)$的顶点集为$V=\{1，2，\ \ldots,n\}$，则哈密顿回路的可能解表示为n元组$X=(x_1,x_2,\ldots,x_n)$，其中，$x_i\in\{1,2,\ldots,n\}$。根据题意可以得到如下的约束条件：

$$\begin{cases}(x_i,x_{i+1})\in E & (1\le i\le n-1)\\ (x_n,x_1)\in E & \\ x_i\neq x_j & (1\le i,j\le n,i\neq j)\end{cases}$$

回溯法求解哈密顿回路问题首先把所有顶点的访问标志初始化为0，然后在解空间树中从根结点开始搜索，如果从根结点到当前结点对应一个部分解，即满足上述约束条件，则在当前结点处选择第一棵子树继续搜索，否则，对当前子树的兄弟结点进行搜索，如果当前结点的所有子树都已尝试过并且发生冲突，则回溯到当前结点的父结点。

算法设计

用回溯法求解TSP回路，首先把n元组$(x_1,x_2,\ldots,x_n)$的每一个分量初始化为0，然后深度优先搜索解空间树，如果满足约束条件累加权值，则继续进行搜索直至结束返回或者直接引起搜索过程矛盾终止提前回溯。设数组$x[n]$存储哈密顿回路上的顶点，数组$vis[n]$存储顶点的访问标志，$vis[i]=1$表示哈密顿回路经过顶点i，算法用伪代码描述如下：

算法：回溯法求解TSP问题
输入：带权有向图$G=(V,E)$
输出：TSP问题最优解
1. 将起点标记，深层遍历并记录访问的结点数量；
2. 每次选择一个未标记的u点进行访问并$vis[u]=1$；回溯后将该顶点再设为未访问状态；

3. 每次到一个新的搜索状态检查是否访问n个顶点，尝试进行答案的更新；
4. 输出TSP最优解答案。

算法分析

实际上TSP问题的回溯本质上还是爆搜遍历，但是更加符合机器的运转形式，可以加入一些优化剪枝操作使运算的实际效果得到显著的提升。

源代码

```
#include <bits/stdc++.h>
#pragma GCC optimize(2)
#pragma G++ optimize(2)
#define endl "\n"
using namespace std;
typedef long long ll;
typedef pair<int, int> pii;
const int maxn = 55;
int n, S;
int dis[maxn][maxn];
int vis[maxn];
int ans = 0x3f3f3f3f;

void dfs(int now, int cnt, int temp) {
    if (temp >= ans) return;  //简单优化
    if (cnt == n)
        if (accumulate(vis + 1, vis + 1 + n, 0) == n) ans = min(ans, temp
 + dis[now][S]);
    for (int i = 1; i <= n; ++i) {
        if (vis[i]) continue;
        vis[i] = 1;
        dfs(i, cnt + 1, temp + dis[now][i]);
        vis[i] = 0;
    }
}

void solve() {
    cin >> n >> S;
    for (int i = 1; i <= n; ++i)
```

```
        for (int j = 1; j <= n; ++j)
            cin >> dis[i][j];
    vis[S] = 1;
    dfs(S, 1, 0);
    cout << ans << endl;
}

signed main() {
    ios::sync_with_stdio(false), cin.tie(0);
#ifdef LOCAL
    freopen("IO\\in.txt", "r", stdin);
    freopen("IO\\out.txt", "w", stdout);
    clock_t start, end;
    start = clock();
#endif
    solve();
#ifdef LOCAL
    end = clock();
    cout << endl
         << "Runtime: " << (double)(end - start)  << "ms\n";
#endif
    return 0;
}
```

算法结果

表 4：TSP 问题回溯法求解结果

城市数量	计算结果	运行时间
5	1033	0ms
10	981	1ms
20	800	9017ms
50	--	$> 30min$

（输入输出文件见附录）

对比蛮力法该算法虽然本质上和上述相同但是运算速度明显加快。相比于蛮力法对于数据量中小范围的数据回溯法能够在可以接受的时限内求出结果，相比于动态规划法其所需要的空间消耗也比较小，能够完成 50 城市以内的求解，相

比于贪心法获得结果完全准确。

分支限界法求解

算法思想

首先确定目标函数的上下界$[down, up]$,可以采用贪心法确定TSP问题的一个上界。对于下界我们采用一个合理的方法保证将答案限制在较小范围内进行有效剪枝，对于有向图的代价矩阵，把矩阵中每一行的最小元素相加，可以得到一个有效的下界。同时考虑一个能够包含更大信息量的下界：考虑一个TSP问题的完整解，在每条路径上每个城市都有两条临接边，一条是进入这个城市的，另一条是离开这个城市的，那么如果把矩阵中每一行最小的两个元素相加再除以 2，假设图中所有的代价都是整数，再对这个结果向上取整，就得到了一个有效的下界。

需要强调的是，这个解可能不是一个可行解，但是给出了一个参考的下界。如果不能确定下界是否合理，其实可以值限制上界进行权值和过大的剪枝。

算法设计

设封装一个结点用来记录代表所处状态的始末点、上界、以及走过的城市，用一个优先队列来记录状态树中每个结点的预测界值并依次进行先后排序，伪代码如下：

算法：分支限界法求解TSP问题
输入：带权有向图$G = (V, E)$
输出：最短哈密顿回路的值
1. 根据选取的限界函数计算目标函数的下界$down$，采用贪心法得到上界up；
2. 计算根结点的目标函数值并将待处理的结点压入优先队列；
3. 循环直到某个叶子结点的目标函数值已经取得最小值
3.1 取出队列顶部的结点i；
3.2 对结点的每个孩子结点x执行下述操作；

3.2.1 利用设定的估算函数估算x的目标函数值lb；

3.2.2 若$lb \leq up$，则将结点x压入队列否则丢弃该结点进行剪枝；

4. 将叶子结点对应的最优值输出，回溯求得最优解的各个分量。

算法分析

算法本质上还是搜索剪枝优化，借助设定合理的边界从而缩小查找的范围以提高计算的速度，但是限界函数的设定以及估算函数的设定直接影响着整个算法的走向，若是不能够有效的限制结点的取值，不仅会徒增了函数计算带来的消耗而且不能得到的有效优化。为了加快计算能力，我们可以在存储结点的取值过程中借助堆优化进行一个简单的改良。

源代码

```
#include <bits/stdc++.h>
#pragma GCC optimize(2)
#pragma G++ optimize(2)
#define endl "\n"
using namespace std;
typedef long long ll;
typedef pair<int, int> pii;
const int maxn = 55;

int n, S;
int dis[maxn][maxn];

struct node {
    bool vis[maxn];                         //标记走过的点
    int start, end;                         //第一个点、最后一个点
    int k, sum, lb;                         //走过的点数、路径距离、目标函数
值
    bool operator<(const node &A) const {  //lb 小的先出队
        return lb < A.lb;
    }
    node() = default;
    node(int start, int end, int k) : start(start), end(end), k(k) { mems
et(vis, 0, sizeof vis); }
```

```
};

priority_queue<node> que;  //创建一个优先队列
int low, up;               //下界和上界
bool dfs_vis[maxn];        //在 dfs 中搜索过

int dfs(int now, int nex, int len) {
    if (nex == n + 1) return len + dis[now][S];
    int minlen = 0x3f3f3f3f, p;
    for (int i = 1; i <= n; ++i)
        if (!dfs_vis[i] && minlen > dis[now][i])
            minlen = dis[now][i], p = i;
    dfs_vis[p] = true;
    return dfs(p, nex + 1, len + minlen);
}

void getup() {  //贪心法确定上界
    dfs_vis[S] = true;
    up = dfs(S, 2, 0);
}

void getlow() {  //确定下界
    double temp = 0;
    int tempdis[maxn];
    for (int i = 1; i <= n; ++i) {
        for (int j = 1; j <= n; ++j)
            tempdis[j] = dis[i][j];
        sort(tempdis + 1, tempdis + 1 + n);
        temp += tempdis[2] + tempdis[3];
    }
    cout << temp << endl;
    low = ceil(temp * 1.0 / 2);
}

int getlb(node p) {
    int temp = p.sum * 2, minn = 0x3f3f3f3f;
    for (int i = 1; i <= n; ++i) {
        if (i == p.start) continue;
        minn = min(minn, dis[p.start][i]);
    }
    temp += minn;
    for (int i = 1; i <= n; ++i) {
        if (i == p.end) continue;
        minn = min(minn, dis[p.end][i]);
```

```
    }
    temp += minn;
    for (int i = 1; i <= n; ++i) {
        if (!p.vis[i]) {
            int tempdis[maxn];
            for (int j = 1; j <= n; ++j)
                tempdis[j] = dis[i][j];
            sort(tempdis + 1, tempdis + 1 + n);
            temp += tempdis[1] + tempdis[2];
        }
    }
    return ceil(temp * 1.0 / 2);
}

void solve() {
    cin >> n >> S;
    for (int i = 1; i <= n; ++i)
        for (int j = 1; j <= n; ++j)
            cin >> dis[i][j];
    getup(), getlow();
    node st(S, S, 1);                              //起点终点走过的点
    st.vis[S] = true, st.sum = 0, st.lb = low;  //初始化标记
    cout << low << ' ' << up << endl;
    int ans = up;  //最终答案
    que.push(st);  //放入队列
    while (que.size()) {
        node now = que.top();
        que.pop();
        if (now.k == n) {  //走过了 n 个点更新答案
            ans = min(ans, now.sum + dis[now.end][now.start]);
            continue;
        }
        for (int i = 1; i <= n; ++i) {
            if (now.vis[i]) continue;
            node nex;
            nex.start = now.start;
            nex.sum = now.sum + dis[now.end][i];
            nex.end = i;
            nex.k = now.k + 1;
            for (int j = 1; j <= n; ++j) nex.vis[j] = now.vis[j];
            nex.vis[i] = 1;
            nex.lb = getlb(nex);
            if (nex.lb > up) continue;  //剪枝
            que.push(nex);
```

```
        }
    }
    cout << ans << endl;
}

signed main() {
    ios::sync_with_stdio(false), cin.tie(0);
#ifdef LOCAL
    freopen("IO\\in.txt", "r", stdin);
    freopen("IO\\out.txt", "w", stdout);
    clock_t start, end;
    start = clock();
#endif
    solve();
#ifdef LOCAL
    end = clock();
    cout << endl
         << "Runtime: " << (double)(end - start) << "ms\n";
#endif
    return 0;
}
```

算法结果

表 5：TSP 问题分支限界法求解结果

城市数量	计算结果	运行时间
5	1033	2ms
10	981	45ms
20	--	$> 1h$
50	--	--

（输入输出文件见附录）

分支限界法对比上述几种算法一个特殊的点就是不稳定，很大取决于设定函数的选取是否合理，这也就意味着其上限是未知的，虽然目前来看其简单的估值函数并不能体现出其良好的性能但是借助今年发展的各种智能预测算法，也许可以将TSP问题带来更多的变革。

0-1 背包问题

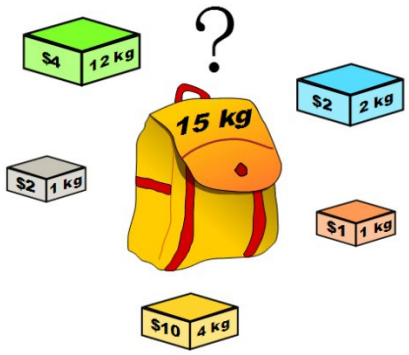

图 2：0-1 背包问题示意图

0-1 背包问题是经典的组合问题，在其基础上产生了多个变种问题，其本质是如何在价值与容量的复杂关系之间寻找最优解。其雏形问题为：给定n个重量为$\{w_1, w_2, \ldots, w_n\}$、价值为$\{v_1, v_2, \ldots, v_n\}$的物品和一个容量为$C$的背包，假定每个物品只能取一次，求这些物品中的一个最有价值的子集，要求能够装到背包中。

蛮力法求解

算法想法

用蛮力法求解 0-1 背包需要考虑给定n个物品集合的所有子集，找出所有总重量不超过背包容量的子集，计算每个可能子集的总价值，然后找到价值最大的子集。

算法设计

我们将n个物品进行全排列，然后每次从容量C依次从左到右按照顺序拿物品，能装得下必须拿，否则跳过。算法伪代码如下：

算法：蛮力法求解 0-1 背包问题
输入：重量$w_1, w_2, \ldots, w_n$，价值$\{v_1, v_2, \ldots, v_n\}$，背包容量为$C$
1. 初始化最大价值$ans = 0$；
2. 对集合$\{1,2, \ldots, n\}$的进行全排列，每次尝试如下操作

2.1 初始化背包剩余容量与得到价值分别为$tempW = C, tempV = 0$；

2.2 对子集的每一个元素j尝试放入背包

2.2.1 如果$w_j < tempW$，则$tempW -= w_j, tempV += v_j$；

2.2.2. 否则转到步骤 2 考察下一个子集

3. 输出自己S中的元素。

算法分析

对于一个具有n个元素的集合，其子集数量是2^n，所以不论生成算法的效率有多高蛮力法求解 0-1 背包问题都会导致时间复杂度为$O(2^n)$。

源代码

```
#include <bits/stdc++.h>
#pragma GCC optimize(2)
#pragma G++ optimize(2)
#define endl "\n"
using namespace std;
typedef long long ll;
typedef pair<int, int> pii;

const int maxn = 55;

int arr[maxn];
int w[maxn], v[maxn];

void solve() {
    int n, W;
    cin >> n >> W;
    for (int i = 1; i <= n; ++i)
        cin >> w[i] >> v[i];
    for (int i = 1; i <= n; ++i) arr[i] = i;
    int ans = 0;
    do {
        int tempW = W, tempV = 0;
        for (int i = 1; i <= n; ++i) {
            if (tempW >= w[arr[i]])
```

```
                tempW -= w[arr[i]], tempV += v[arr[i]];
            else
                break;
        }
        ans = max(ans, tempV);
    } while (next_permutation(arr + 1, arr + 1 + n));
    cout << ans << endl;
}

signed main() {
    ios::sync_with_stdio(false), cin.tie(0);
#ifdef LOCAL
    freopen("IO\\in.txt", "r", stdin);
    freopen("IO\\out.txt", "w", stdout);
    clock_t start, end;
    start = clock();
#endif
    solve();
#ifdef LOCAL
    end = clock();
    cout << endl
         << "Runtime: " << (double)(end - start) << "ms\n";
#endif
    return 0;
}
```

算法结果

表 6：0-1 背包问题蛮力法求解结果

物品数量	计算结果	运行时间
5	468	1ms
10	505	161ms
20	--	$> 10min$
50	--	--

（输入输出文件见附录）

对于组合问题来说，穷举试探是一种最简单的蛮力方法，也就是依次生成并考察解空间中的每一个组合对象，从中选出满足问题约束的解。对组合问题应用

蛮力法，除非问题规模很小，否则会由于问题的解空间太大而产生的组合爆炸的问题。

动态规划法求解

算法想法

首先采用反证法证明 0-1 背包问题满足最优性原理。设$(x_1, x_2, \ldots, x_n)$是 0-1 背包问题的最优解，则$(x_2, \ldots, x_n)$是下面子问题的最优解：

$$\begin{cases} \sum_{i=2}^{n} w_i x_i \le C - w_1 x_1 \\ x_i \in \{0,1\} (2 \le i \le n) \end{cases}, \text{求} max \sum_{i=2}^{n} v_i x_i$$

如果不然，设$(y_2, \ldots, y_n)$是 0-1 背包的最优解，则$\sum_{i=2}^{n} v_i y_i > \sum_{i=2}^{n} v_i x_i$，且$w_1 x_1 + \sum_{i=2}^{n} w_i y_i \le C$。因此，$v_1 x_1 + \sum_{i=2}^{n} v_i y_i > v_1 x_1 + \sum_{i=2}^{n} v_i x_i$，这说明$(x_1, y_2, \ldots, y_n)$是 0-1 背包的最优解且比$(x_1, x_2, \ldots, x_n)$更优，从而导致矛盾。我们定义用二维表示的子问题$V(n, C)$表示将$n$个物品装入容量为$C$的背包获得的最大价值，显然初始子问题是把前面$i$个物品装入容量为 0 的背包和把 0 个物品装入容量为j的背包得到价值均为 0，即：

$$V(i, 0) = V(i, j) = 0 \; 0 \le i \le n, 0 \le j \le C$$

考虑将原问题的一部分，设$V(i, j)$表示将前$i(1 \le i \le n)$）个物品装入容量为$j(1 \le j \le C)$的背包获得的最大价值，在决策x_i时，已经确定了$(x_1, \ldots, x_{i-1})$,则问题处于下列两种状态之一：

1） 背包容量不足以装入物品i，则装入前i个物品得到的最大价值和装入前 i-1 个物品得到的最大价值是相同的，即$x_i = 0$背包不增加价值。

2）背包容量可以装入物品i，如果把第i个物品装入背包，则背包中物品的价值等于把前$i-1$个物品装入容量为$j - w_i$的背包中的价值加上第i个物品的价值v_i；如果第i个物品没有装入背包，则背包中物品的价值等于把前$i-1$个物品装入容量为j的背包中所取得的价值。显然，取二者中价值较大者作为把前i个物品装入容量为j的背包中的最优解。则得到如下递推方程式：

$$V(i, j) = \begin{cases} V(i-1, j) & j < w_i \\ max\{V(i-1, j), V(i-1, j-w_i) + v_i\} & j \ge w_i \end{cases}$$

为了确定装入背包的具体物品，从$V(n, C)$的值向前推，如果$V(n, C) > V(n -$

$1, C)$, 表明第n个物品被装入背包，前$n-1$个物品被装入容量为$C-w_i$的背包中；否则第n个物品没有被装入背包，前$n-1$个物品被装入容量为C的背包中。依次类推，直到确定第 1 个物品是否被装入背包中为止。由此得到如下函数：

$$x_i = \begin{cases} 0 & V(i,j) = V(i-1,j) \\ 1 & v(i,j) > V(i-1,j) \end{cases}$$

算法设计

设n个物品的重量存储在数组$w[]$中，价值存储在$v[]$中，背包容量为W，通过滚动降维开$dp[]$记录上述算法想法中的V第二维，表示循环每一轮（即放入当前物品）容量为j的背包获得的最大价值，算法的伪代码如下：

算法：动态规划法求解 0-1 背包
输入：物品个数n，背包容量C，n个物品的价值和重量
1. 初始化边界，未装入物品（即考虑装入第 0 个物品）时对于所有容量的背包$dp[j] = 0$；
2. 循环n轮循环（即考虑装入$1 \sim n$每个物品）求解所有可能容量情况下的最优解
2.1 如果物品装不下则不用修改$dp[j]$继续下一轮循环；
2.1 否则考虑装与不装两种情况的最优方法并更新$dp[j]$；
3. 输出答案$dp[W]$。

算法分析

由于本质还是考虑所有情况的最优解类似于暴力，随着问题规模的增大空间消耗将明显增大借助滚动降维减少表示每个状态占用的内存可以提高算法的适用范围。另一方面也可以进行一些时间上的优化，借助前缀和减少每个物品需要转移的次数。

整体来看算法主要性能由两重循环确定，第一重for循环性能为$O(n)$，第二重for循环时间性能不考虑常数优化为$O(C)$，总的时间性能为$O(nC)$。

源代码

```
#include <bits/stdc++.h>
#pragma GCC optimize(2)
#pragma G++ optimize(2)
#define debug(x) cout << "debug: " << x << endl;
using namespace std;
typedef long long ll;
const int maxn = 1e3 + 5;

int v[maxn], w[maxn], dp[maxn], sum[maxn];

void solve() {
    int n, W;
    cin >> n >> W;
    for (int i = 1; i <= n; ++i) cin >> w[i] >> v[i], sum[i] = sum[i - 1]
 + v[i];
    for (int i = 1; i <= n; ++i) {
        int bound = max(w[i], W - (sum[n] - sum[i]));  //前缀和处理
        for (int j = W; j >= bound; --j)
            dp[j] = max(dp[j], dp[j - w[i]] + v[i]);
    }
    cout << dp[W] << endl;
}

signed main() {
    ios::sync_with_stdio(false), cin.tie(0);
#ifdef LOCAL
    freopen("IO\\in.txt", "r", stdin);
    freopen("IO\\out.txt", "w", stdout);
    clock_t start, end;
    start = clock();
#endif
    solve();
#ifdef LOCAL
    end = clock();
    cout << endl
         << "Runtime: " << (double)(end - start) << "ms\n";
#endif
    return 0;
}
```

算法结果

表 7：0-1 背包问题蛮动态规划法求解结果

物品数量	计算结果	运行时间
5	468	0ms
10	505	0ms
20	709	0ms
50	2113	0ms

（输入输出文件见附录）

相比于蛮力法动态规划的时间复杂度明显更加高效，体现出了经典的空间换时间思想。

贪心法求解

算法想法

用贪心法求解背包问题的关键是如何选定贪心策略，使得按照一定的顺序选择每个物品，并尽可能的装入背包，直到背包装满。至少有三种看似合理的贪心策略：

1）选择价值最大的物品，因为这可以尽可能快的增加背包的总价值。但是，虽然每一步选择获得了背包价值的极大增长，但背包容量却可能消耗得太快，使得装入背包的物品个数减少，从而不能保证目标函数达到最大。

2）选择重量最轻的物品，因为这可以装入尽可能多的物品，从而增加背包的总价值。但是虽然每一步选择使背包的容量消耗得慢了，但背包的价值却没能保证迅速增长，从而不能保证目标函数达到最大。

3）以上两种贪心策略或者只考虑背包价值的增长，或者只考虑背包容量的消耗，而为了求得背包问题的最优解，需要在背包价值增长和背包容量消耗二者之间寻找平衡。正确的贪心策略是选择单位重量价值最大的物品。

算法设计

设背包容量为C，共有n个物品，物品重量存放在数组$w[]$中，价值存放在数组$v[]$中，采用上述三重贪心策略选取最优作为答案，算法伪代码如下：

算法：贪心法求解 0-1 背包问题

输入：背包容量W，物品重量$w[]$，物品价值$v[]$

输出：贪心策略求解出的最优解

1. 依次进行按照物品价值、物品重量、物品单位重量价值进行排序；
2. 每次排序后一次执行如下操作
   2.1 从左向右依次查看每个物品，如果能拿则必须拿，否则跳过；
   2.2 答案记录当前策略下获取的价值$tempV$；
3. 更新答案并继续步骤 1；
4. 输出最终答案。

算法分析

时间主要消耗在将各种物品进行排序，因此时间复杂度为$O(nlogn)$。背包问题与 0-1 背包问题类似，所不同的是在选择物品$i(1 \le i \le n)$装入背包时可以选择一部分，而不一定要全部装背包。背包问题可以用贪心问题求解，而 0-1 背包问题却不能用贪心法求解。因为 0-1 背包问题存在重叠子问题，存在后效性所以最好的做法应该是动态规划法求解。

源代码

```
#include <bits/stdc++.h>
#pragma GCC optimize(2)
#pragma G++ optimize(2)
#define endl "\n"
using namespace std;
typedef long long ll;
typedef pair<int, int> pii;
const int maxn = 55;
```

```
struct thing {
    int weight, value, id;
} beg[maxn];

bool sortByValue(thing A, thing B) {  //物品价值大的优先
    return A.value > B.value;
}

bool sortByWeight(thing A, thing B) {  //物品重量轻的优先
    return A.weight < B.weight;
}

bool sortByRate(thing A, thing B) {  //物品价值重量比大的优先
    return A.value * 1.0 / A.weight > B.value * 1.0 / B.weight;
}

void solve() {
    int n, W;
    cin >> n >> W;
    for (int i = 1; i <= n; ++i)
        cin >> beg[i].weight;
    for (int i = 1; i <= n; ++i)
        cin >> beg[i].value, beg[i].id = i;

    int ans = 0;
    sort(beg + 1, beg + n + 1, sortByValue);
    int tempW = W, tempV = 0;
    for (int i = 1; i <= n; ++i) {
        if (beg[i].weight <= tempW)
            tempW -= beg[i].weight, tempV += beg[i].value;
    }
    ans = max(ans, tempV);

    tempW = W, tempV = 0;
    sort(beg + 1, beg + n + 1, sortByWeight);
    for (int i = 1; i <= n; ++i) {
        if (beg[i].weight <= tempW)
            tempW -= beg[i].weight, tempV += beg[i].value;
    }
    ans = max(ans, tempV);

    tempW = W, tempV = 0;
    sort(beg + 1, beg + n + 1, sortByRate);
```

```
    for (int i = 1; i <= n; ++i) {
        if (beg[i].weight <= tempW)
            tempW -= beg[i].weight, tempV += beg[i].value;
    }
    ans = max(ans, tempV);
    cout << ans << endl;
}

signed main() {
    ios::sync_with_stdio(false), cin.tie(0);
#ifdef LOCAL
    freopen("IO\\in.txt", "r", stdin);
    freopen("IO\\out.txt", "w", stdout);
    clock_t start, end;
    start = clock();
#endif
    solve();
#ifdef LOCAL
    end = clock();
    cout << endl
         << "Runtime: " << (double)(end - start) << "ms\n";
#endif
    return 0;
}
```

算法结果

表 8：0-1 背包问题贪心法求解结果

物品数量	计算结果	运行时间
5	260	0ms
10	147	0ms
20	326	0ms
50	822	0ms

（输入输出文件见附录）

回溯法求解

算法想法

仍然利用动态规划法求解的思想考虑每个物品为一个结点，那么这个节点在可以放入背包的情况有选与不选两种可能。如此可以看成$1\sim n$个物品形成一个满二叉树，每个从树根到叶子节点路径上经过的边累加就形成了一个策略。我们想要最优解则可以搜索每种策略从而得到最优解，在搜索过程中我们可以设定上界函数对于无法成为最优解的策略进行剪枝。通过回溯我们可以在一遍树的搜索遍历中找到最优解的策略。

算法设计

对于每个物品结点我们分别记录从上个物品结点搜索过来时背包剩余的容量与已经装入物品的总价值。算法伪代码如下：

算法：回溯法求解 0-1 背包问题
输入：背包容量W，物品重量$w[]$，物品价值$v[]$
输出：贪心策略求解出的最优解
1. 从第一个物品开始进行考虑装或者不装两种选择
1.1 如果物品无法装入当前背包容量则返回步骤一；否则尝试两种选择产生分支；
1.2 如果物品结点深度达到n则走完一个完整的策略，用当前的$tempV$更新答案，回溯到上一个物品结点，如果存在分支则尝试下一种；
2. 当第一个物品出发的所有策略都搜索结束则输出被更新的答案

算法分析

回溯法本质仍然是暴力但是通过在递归搜索过程中设置合理的上界函数可以进行有效的剪枝进行优化，时间性能取决于函数设定的合理度。

源代码

```
#include <bits/stdc++.h>
#pragma GCC optimize(2)
#pragma G++ optimize(2)
#define endl "\n"
#define fi first
#define se second
#define pb push_back
#define all(x) x.begin(), x.end()
#define rep(i, x, y) for (auto i = (x); i != (y + 1); ++i)
#define dep(i, x, y) for (auto i = (x); i != (y - 1); --i)
#ifdef LOCAL
#define de(...) cout << '[' << #__VA_ARGS__ << "] = " << __VA_ARGS__ << e
ndl;
#else
#define de(...)
#endif
using namespace std;
typedef long long ll;
typedef pair<int, int> pii;
const int maxn = 55;

int n, W;
int w[maxn], v[maxn];

int dfs(int now, int tempW, int tempV) {
    if (now == n + 1) return 0;
    int ans = tempV;
    if (w[now] <= tempW) ans = max(ans, dfs(now + 1, tempW - w[now], temp
V + v[now]));
    ans = max(ans, dfs(now + 1, tempW, tempV));
    return ans;
}

void solve() {
    cin >> n >> W;
    for (int i = 1; i <= n; ++i) cin >> w[i] >> v[i];
    cout << dfs(1, W, 0) << endl;
}

signed main() {
    ios::sync_with_stdio(false), cin.tie(0);
```

```
#ifdef LOCAL
    freopen("IO\\in.txt", "r", stdin);
    freopen("IO\\out.txt", "w", stdout);
    clock_t start, end;
    start = clock();
#endif
    solve();
#ifdef LOCAL
    end = clock();
    cout << endl
         << "Runtime: " << (double)(end - start) << "ms\n";
#endif
    return 0;
}
```

算法结果

表 9：0-1 背包问题回溯法求解结果

物品数量	计算结果	运行时间
5	460	1ms
10	505	1ms
20	709	2ms
50	2113	15075ms

（输入输出文件见附录）

由于回溯法是在树上进行搜索，随着物品数量规模的增大，其二叉树的结点数量成指数式增加，往后所需要的计算代价也逐渐增大，不设置有效的剪枝无法做到时间的减少，还有可能由于大量递归的传参拷贝造成时间性能的下降。

分支限界法

算法想法

假设n种物品已按单位价值由大到小排序，这样第一个物品给出了单位重量的最大价值，最后一个物品给出了单位重量的最小价值。可以采用贪心法求得 0-

1 背包问题的一个下界。考虑上界可以假设最好的情况，背包装入的全部都是第一个物品且可以将背包装满，则可以得到一个非常简单的计算方法。$ub = W * (v1/w_1)$。一般情况下，假设当前已对前i个物品进行了某种特定的选择，且背包中已装入物品的重量为w，获得的价值为v，计算该结点的目标函数上界的一个简单方法是将背包中剩余容量全部转给第$i+1$个物品，并可以将背包装满，于是，得到限界函数：

$$ub = v + (W - w) * (v_{i+1}/w_{i+1})$$

算法设计

设背包容量为C，共有n个物品，物品重量存放在数组$w[]$中，价值存放在数组$v[]$中，算法伪代码如下：

算法：分支限界法求解 0-1 背包问题

输入：n个物品的重量$w[]$，价值$v[]$，背包容量W

1. 根据限界函数计算目标函数的上界up；采用贪心法得到下界；
2. 计算根结点的目标函数值并加入优先队列中；
3. 循环直到某个叶子节点的目标函数值在表中取得极大值
   3.1 i =优先队列中具有最大值的结点；
   3.2 对结点i的每个孩子结点x执行下列操作
      3.2.1 如果结点x不满足约束条件，则丢弃该结点；
      3.2.2 否则估算结点x的目标函数值lb，将结点x加入优先队列中；
4. 将叶子结点对应的最优值输出，回溯求得最优解的各个量。

算法分析

算法本质上还是搜索剪枝优化，借助设定合理的边界从而缩小查找的范围以提高计算的速度，但是限界函数的设定以及估算函数的设定直接影响着整个算法的走向，若是不能够有效的限制结点的取值，不仅会徒增了函数计算带来的消耗而且不能得到的有效优化。为了加快计算能力，我们可以在存储结点的取值过程中借助堆优化进行一个简单的改良。

源代码

```
#include <bits/stdc++.h>
#pragma GCC optimize(2)
#pragma G++ optimize(2)
#define endl "\n"
using namespace std;
typedef long long ll;
typedef pair<int, int> pii;
const int maxn = 55;

int lb, ub;
pii a[maxn];

struct node {
    int w, v, bound, step;
    bool operator<(const node &A) const {
        return this->bound > A.bound;
    }
    node(int w, int v, int bound, int step) : w(w), v(v), bound(bound), s
tep(step) {}
};

void solve() {
    int n, W;
    cin >> n >> W;
    for (int i = 1; i <= n; ++i)
        cin >> a[i].first >> a[i].second;
    sort(a + 1, a + 1 + n, [=](pii A, pii B) {
        return A.second * 1.0 / A.first > B.second * 1.0 / B.first;
    });
    int tmp = W;
    for (int i = 1; i <= n; ++i)
        if (tmp >= a[i].first) tmp -= a[i].first, lb += a[i].second;
    tmp = W;
    ub = a[1].second * 1.0 / a[1].first * tmp;
    priority_queue<node> que;
    que.push(node(0, 0, 0, 0));
    int ans = 0;
    while (que.size()) {
        auto now = que.top();
        ans = max(ans, now.v);
        que.pop();
```

```
        auto nex = now;
        nex.step = now.step + 1;
        if (now.step == n + 1) continue;
        //先考虑不装进去
        auto tmp = nex;
        if (W - now.w >= a[tmp.step].first) {
            tmp.w = tmp.w + a[tmp.step].first;
            tmp.v = tmp.v + a[tmp.step].second;
            if (tmp.step != n)
                tmp.bound = tmp.v + (W - tmp.w) * a[tmp.step + 1].second
* 1.0 / a[tmp.step + 1].first;
            if (tmp.bound >= lb && tmp.bound <= ub) que.push(tmp);
        }
        tmp = nex;
        if (tmp.step != n)
            tmp.bound = tmp.v + (W - tmp.w) * a[tmp.step + 1].second * 1.
0 / a[tmp.step + 1].first;
        if (tmp.bound >= lb && tmp.bound <= ub) que.push(tmp);
    }
    cout << ans << endl;
}

signed main() {
    ios::sync_with_stdio(false), cin.tie(0);
#ifdef LOCAL
    freopen("IO\\in.txt", "r", stdin);
    freopen("IO\\out.txt", "w", stdout);
    clock_t start, end;
    start = clock();
#endif
    solve();
#ifdef LOCAL
    end = clock();
    cout << endl
         << "Runtime: " << (double)(end - start) << "ms\n";
#endif
    return 0;
}
```

算法结果

表 10：0-1 背包问题分支限界法求解结果

物品数量	计算结果	运行时间
5	460	0ms
10	505	1ms
20	709	1ms
50	2113	1ms

（输入输出文件见附录）

从结果来看该分支限界法对于我所造的样例比较适用，但是其实时间效率非常不稳定，往往限界函数设定的不合理可能导致计算时间没有太多的降低，经过洛谷*OJ*测试对于超过1000个物品时该算法的计算消耗显著提高远远落后于动态规划法。

算法对比分析

表 1：五种算法定性比较

指标	蛮力法	动态规划法	贪心法	回溯法	分支限界法
时间复杂度	极高	低	极低	高	中
空间复杂度	低	极高	极低	中	高
是否准确	是	是	否	是	是
实现代价	极低	高	低	中	极高
回溯路径	极易	极难	易	中	难
未来发展	无	硬件改良	策略优化	硬件改良	算法改良

sudoku问题

数独问题是一款对人智慧和毅力考验的益智游戏。它需要人利用自己所有的想象力、逻辑推理和创新思维预测正确的答案在尽可能短时间内根据$9*9$盘面上的已知数字推理出所有剩余空格的数字，并满足每一行每一列，每一个出现宫内中的数字均包含1~9,且不重复。

	6						7	5
				4			9	
8		3						
		8		9				
	7	2			6			
6			4		5			
1	2		6					
	3			7		2		
		7			4	3	1	

图 3：sudoku 问题示意图

如今借助计算力强大的计算机，我们设计合理的算法指引计算机去尝试各种可能的组合答案即可将一个看起来对人而言复杂的经典数独问题在毫厘的时间内求解出来。

其中，最简单也是最暴力的一种方法便是借助回溯法的基础上加上剪枝优化求解。

算法思想

我们从左上角的第一个位置开始自左向右、自上向下尝试给每个位置放1~9数字，每次放下尝试的数字后检查是否和之前位置上的数存在冲突，如果确实存在则提前返回否则就继续给下一个位置放数字，每次完成该操作后回溯到上一个位置进行下一个数字的尝试直到放完所有的数字说明完成了数独题目的求解。

算法设计

假设我们用$num[][]$存储题目给出的$9*9$棋盘，行列下标从 0 开始，我们从第 0 行第 0 列的位置开始尝试在每个空的位置放1~9数字，每次尝试放完当前位置的数字后进行矛盾检查，即其所处的行方向、列方向、以及所处的九宫格位置上没有与其相同的数字。如果存在矛盾说明当前位置可以进行剪枝优化直接跳过深度的搜索尝试放下一个数字，否则继续深度搜索返回时进行回溯后再尝试放下一个数字。由于经典的数独问题保证只有一个正解，所以当某次循环搜到第 81 个位置时说明找到了正解直接退出。算法伪代码如下：

算法：回溯法求解数独问题

输入：9 * 9棋盘

输出：填完所有数字的完整棋盘

1. 存入棋盘，将没有数字的位置设为 0；
2. 从第一个位置开始尝试给每个空位置放置 1~9数字直至找到正解
    2.1 检查是否放完 81 个位置，如若是直接终止搜索；
    2.2 当前位置存在数字直接跳过；
    2.3 否则，当前位置数字为空尝试从1~9放置数字
        2.3.1 检查当前位置是否存在冲突；
        2.3.2 存在冲突回到步骤 2.1；
        2.3.3 否则不存在冲突继续处理下一个位置；
    2.4 清空当前位置放置的数字回溯当上一层；
3. 输出填好的棋盘。

算法分析

经典的数独问题数据很小，以目前小型计算机的运算能力都足以快速的求解。该回溯法不进行剪枝也足以短时间内完成求解，关键在于该方法的设计值得我们学习，它通过模仿我们的常规认识对每个位置数字的局部寻找矛盾从而进行有效的剪枝。

源代码

```
#include <bits/stdc++.h>
using namespace std;
/* 构造完成标志 */
bool sign = false;
/* 创建数独矩阵 */
int num[9][9];
/* 读入数独矩阵 */
void Input() {
    char temp[9][9];
```

```
    for (int i = 0; i < 9; i++) {
        for (int j = 0; j < 9; j++) {
            cin >> temp[i][j];
            num[i][j] = temp[i][j] - '0';
        }
    }
}
/* 输出数独矩阵 */
void Output() {
    for (int i = 0; i < 9; i++) {
        for (int j = 0; j < 9; j++) {
            cout << num[i][j] << " ";
            if (j % 3 == 2) {
                cout << "   ";
            }
        }
        cout << endl;
        if (i % 3 == 2) {
            cout << endl;
        }
    }
}
/* 判断 key 填入 n 时是否满足条件 */
bool Check(int n, int key) {  //寻找冲突
    /* 判断 n 所在横列是否合法 */
    for (int i = 0; i < 9; i++) {
        /* j 为 n 竖坐标 */
        int j = n / 9;
        if (num[j][i] == key) return false;
    }
    /* 判断 n 所在竖列是否合法 */
    for (int i = 0; i < 9; i++) {
        /* j 为 n 横坐标 */
        int j = n % 9;
        if (num[i][j] == key) return false;
    }
    /* x 为 n 所在的小九宫格左顶点竖坐标 */
    int x = n / 9 / 3 * 3;
    /* y 为 n 所在的小九宫格左顶点横坐标 */
    int y = n % 9 / 3 * 3;
    /* 判断 n 所在的小九宫格是否合法 */
    for (int i = x; i < x + 3; i++) {
        for (int j = y; j < y + 3; j++) {
            if (num[i][j] == key) return false;
```

```
        }
    }
    /* 全部合法，返回正确 */
    return true;
}
/* 深搜构造数独 */
void DFS(int n) {
    /* 所有的都符合，退出递归 */
    if (n > 80) {
        sign = true;
        return;
    }
    /* 当前位不为空时跳过 */
    if (num[n / 9][n % 9] != 0) {
        DFS(n + 1);
    } else {
        /* 否则对当前位进行枚举测试 */
        for (int i = 1; i <= 9; i++) {
            /* 满足条件时填入数字 */
            if (Check(n, i) == true) {
                num[n / 9][n % 9] = i;
                /* 继续搜索 */
                DFS(n + 1);
                /* 返回时如果构造成功，则直接退出 */
                if (sign == true) return;
                /* 如果构造不成功，还原当前位 */
                num[n / 9][n % 9] = 0;
            }
        }
    }
}
/* 主函数 */
signed main() {
    ios::sync_with_stdio(false), cin.tie(0);
#ifdef LOCAL
    freopen("IO\\in.txt", "r", stdin);
    freopen("IO\\out.txt", "w", stdout);
    clock_t start, end;
    start = clock();
#endif
    Input();
    DFS(0);
    Output();
#ifdef LOCAL
```

```
    end = clock();
    cout << endl
         << "Runtime: " << (double)(end - start) << "s\n";
#endif
    return 0;
}
```

致谢

岁月如梭，时光荏苒，上学期数据结构课程讲堂的记忆仍历历在目，转眼间这学期的算法课也在欢声笑语中迎来尾声。程序设计对我个人来说贯穿了整个大学生活，从大一进入校 ACM 队的新奇到如今正式修完算法与数据结构这门专业课，这两门课的知识我用了近乎两年的时间学习也只是刚刚步入门槛，其知识之广之深之杂超过任何一门课程的内容，在课堂中以及比赛中我看到了许多许多热爱算法且实力强劲的同龄人，识到了人外有人，天外有天。

我一直以来没有系统的了解过算法的体系架构，在我的印象中每个算法相互独立，通过这门课程的学习与上机实验我体会到了不同算法之间的区别与联系，学会了如何从零开始思考，完整的设计出一个算法并分析其复杂度，学习的过程是痛苦乏味的，但是收获知识带来的快乐与热爱一次又一次鼓励着我成功的坚持了下来。

感谢季晓慧老师为我带来的每一节精彩的算法课堂，很荣幸能够在我短暂的求学生涯中遇到了您为我指引方向，最后也致敬那个日夜勤奋的自己，希望未来的我能够坚定的走完自己选择的道路。

参考文献

王红梅, 胡明. 算法设计与分析[M]. 2 版. 北京:清华大学出版社, 2013.4.

声明

部分图片与文字来源于网络

附录

*TSP*问题随机数据源代码

```
#include <bits/stdc++.h>
#pragma GCC optimize(2)
#pragma G++ optimize(2)
#define endl "\n"
using namespace std;

void solve() {
    int n = 50, m = 1;
    cout << n << ' ' << m << endl;
    for (int i = 1; i <= n; ++i) {
        for (int j = 1; j <= n; ++j)
            cout << (i == j ? 0 : 1 + rand() % 500) << ' ';
        cout << endl;
    }
}

signed main() {
    ios::sync_with_stdio(false), cin.tie(0);
    freopen("IO\\in.txt", "r", stdin);
    freopen("IO\\out.txt", "w", stdout);
    clock_t start, end;
    return 0;
}
```

*TSP*问题 5 城市数据

```
5 1
0 42 468 335 1
170 0 225 479 359
463 465 0 206 146
282 328 462 0 492
496 443 328 437 0
```

*TSP*问题 10 城市数据

```
10 1
0 42 468 335 1 170 225 479 359 463
465 0 206 146 282 328 462 492 496 443
328 437 0 392 105 403 154 293 383 422
217 219 396 0 448 227 272 39 370 413
```

168 300 36 395 0 204 312 323 334 174
165 142 212 254 369 0 48 145 163 258
38 360 224 242 30 279 0 317 36 191
343 289 107 41 443 265 149 0 447 306
391 230 371 351 7 102 394 49 0 130
124 85 455 257 341 467 377 432 309 0

TSP问题 20 城市数据

20 1
0 42 468 335 1 170 225 479 359 463 465 206 146 282 328 462 492 496 443 32
8
437 0 392 105 403 154 293 383 422 217 219 396 448 227 272 39 370 413 168
300
36 395 0 204 312 323 334 174 165 142 212 254 369 48 145 163 258 38 360 22
4
242 30 279 0 317 36 191 343 289 107 41 443 265 149 447 306 391 230 371 35
1
7 102 394 49 0 130 124 85 455 257 341 467 377 432 309 445 440 127 324 38
39 119 83 430 42 0 334 116 140 159 205 431 478 307 174 387 22 246 425 73
271 330 278 74 98 13 0 487 291 162 137 356 268 156 75 32 53 351 151 442
225 467 431 108 192 8 338 0 458 288 254 384 446 410 210 259 222 89 423 44
7
7 31 414 169 401 92 263 156 0 411 360 125 38 49 484 96 42 103 351 292
337 375 21 97 22 349 200 169 485 0 282 235 54 500 419 439 401 289 128 468

229 394 149 484 308 422 311 118 314 15 0 310 117 436 452 101 250 20 57 29
9
304 225 9 345 110 490 203 196 486 94 344 0 24 88 315 4 449 201 459 119
81 297 299 282 90 299 10 158 473 123 39 293 0 39 180 191 158 459 192 316
389 157 12 203 135 273 56 329 147 363 387 376 434 0 370 143 345 417 382 4
99
323 152 22 200 58 477 393 390 76 213 101 11 4 370 0 362 189 402 290 256
424 3 86 183 286 89 427 118 258 333 433 170 155 222 190 0 477 330 369 193

426 56 435 50 442 13 146 61 219 254 140 424 280 497 188 30 0 50 438 367
450 194 196 298 417 287 106 489 283 456 235 115 202 317 172 287 264 0 314
356
186 54 413 309 333 446 314 257 322 59 147 483 482 145 197 223 130 162 0 3
6
451 174 467 45 160 293 440 254 25 155 11 246 150 187 314 475 23 169 19 0

TSP问题 50 城市数据
50 1 0 42 468 335 1 170 225 479 359 463 465 206 146 282 328 462 492 496 443 32 8 437 392 105 403 154 293 383 422 217 219 396 448 227 272 39 370 413 168 300 36 395 204 312 323 334 174 165 142 212 254 369 0 48 145 163 258 38 360 224 242 30 279 317 36 191 343 289 107 41 443 265 149 447 306 391 230 371 351 7 102 394 49 130 124 85 455 257 341 467 3 77 432 309 445 440 127 324 38 39 119 83 430 42 0 334 116 140 159 205 431 478 307 174 387 22 246 425 73 271 330 27 8 74 98 13 487 291 162 137 356 268 156 75 32 53 351 151 442 225 467 431 1 08 192 8 338 458 288 254 384 446 410 210 259 222 89 0 423 447 7 31 414 169 401 92 263 156 411 360 125 38 49 484 96 42 103 351 292 337 375 21 97 22 349 200 169 485 282 235 54 500 419 439 4 01 289 128 468 229 394 149 484 308 422 311 118 314 15 0 310 117 436 452 101 250 20 57 299 304 225 9 345 110 490 203 196 486 94 344 24 88 315 4 449 201 459 119 81 297 299 282 90 299 10 1 58 473 123 39 293 39 180 191 158 459 192 316 389 157 12 0 203 135 273 56 329 147 363 387 376 434 370 143 345 4 17 382 499 323 152 22 200 58 477 393 390 76 213 101 11 4 370 362 189 402 290 256 424 3 86 183 286 89 427 118 258 333 433 170 155 222 190 0 477 330 369 193 426 56 435 50 442 13 146 61 219 254 140 424 280 497 188 30 50 438 367 450 194 196 298 417 287 106 489 28 3 456 235 115 202 317 172 287 264 314 356 186 54 413 309 333 446 314 257 0 322 59 147 483 482 145 197 223 130 162 36 45 1 174 467 45 160 293 440 254 25 155 11 246 150 187 314 475 23 169 19 288 406 459 392 203 126 478 415 315 325 335 375 373 160 334 71 488 298 19 178 0 274 271 264 169 193 486 103 481 214 128 3 03 100 28 126 44 425 24 473 62 182 4 433 6 94 226 32 493 143 223 287 65 4 01 188 361 414 475 271 171 236 334 212 261 397 168 286 51 141 195 196 125 0 20 126 77 195 159 303 372 467 179 94 352 485 19 465 120 153 301 88 61 427 11 258 171 316 77 228 44 259 165 11 0 383 87 66 488 78 475 126 128 130 429 424 21 403 463 124 97 238 262 196 26 0 265 261 203 117 31 327 12 272 412 48 154 21 291 425 189 264 441 352 163 330 401 214 459 79 366 8 478 201 59 440 304 261 358 325 478 109 114 388 302 351 461 429 494 385 406 41 112 205 336 357 0 73 351 324 486 57 217 127 35 8 27 358 338 272 370 362 397 23 118 113 218 197 86 42 424 130 230 66 60 4 33 297 356 54 463 85 235 155 473 458 370 33 464 108 484 412 136 68 349 176 439 224 143 0 255 12 242 176 460 326 22 2 371 127 435 206 284 351 399 280 202 194 235 138 35 57 494 177 206 463 4 9 382 301 414 142 356 356 143 463 112 378 425 179 253 444 297 174 41 314 376 73 319 111 18 433 0 113 196 170 332 41 489 186 91 498 90 491 146 354 315 152 241 45 259 336 260 193 106 265 182 4 3 30 276 109 293 498 50 57 62 128 468 42

130 241 314 175 102 78 216 184 214 493 325 102 393 260 0 171 429 28 85 76 287 499 471 288 348 105 4 222 164 207 364 11 172 490 241 165 43 120 414 92 205 319 233 251 206 476 40 304 423 99
248 85 149 472 365 414 76 46 213 47 179 270 263 20 486 0 290 445 366 41 246 9 319 371 102 324 133 473 153 88 71 264 402 104 424 28 101 470 16 66 29 44 348 89 444 138 410 464 50 182
89 343 109 61 222 259 455 389 147 191 450 344 431 121 249 68 0 37 284 36 227 186 39 354 130 225 249 424 360 258 267 445 456 319 227 412 26 356 2 50 497 85 16 465 343 76 414 143 197 449
73 427 107 174 430 405 206 127 313 376 94 66 37 237 142 315 495 0 257 153 437 339 483 356 16 132 231 342 126 12 138 187 191 151 163 135 394 354 417 453 9 263 234 455 304 135 304 257 149 125
318 214 110 29 201 81 319 359 51 156 362 265 404 177 144 410 403 62 0 490 449 283 154 175 221 403 424 332 370 379 260 9 120 472 4 446 282 5 393 186 314 199 90 223 439 38 411 462 235 9
462 460 494 16 270 438 370 59 201 472 265 118 216 56 316 331 40 213 289 0 83 455 86 211 485 275 381 316 452 42 116 180 111 399 74 289 478 133 457 190 114 9 442 291 224 364 29 185 279 201
72 386 475 72 334 368 154 296 169 326 177 130 151 99 310 194 187 81 117 250 0 168 29 180 365 422 406 327 317 17 227 167 88 182 465 341 187 22 163 222 65 310 416 403 374 125 442 246 263 424
32 307 269 319 103 408 308 482 13 137 131 115 310 85 57 291 294 497 153 55 346 0 209 249 492 213 132 115 440 459 223 205 496 53 270 480 239 424 419 367 160 499 487 197 463 134 159 23 147 393
38 426 148 459 103 308 99 331 293 101 279 300 353 449 383 41 316 76 263 68 337 398 0 419 398 329 352 317 231 450 426 159 230 21 441 61 148 163 156 176 293 362 255 399 147 215 447 189 70 139
164 76 16 22 476 116 29 235 71 406 465 58 463 162 25 50 470 331 424 351 334 426 411 0 238 337 338 279 394 137 215 165 92 450 136 6 338 5 338 124 165 471 109 69 282 86 153 374 153 195
377 327 397 73 250 141 175 320 444 112 442 290 420 66 306 86 217 451 116 110 65 167 394 75 0 10 301 196 74 90 162 173 469 359 32 269 427 11 423 275 280 411 53 183 392 496 265 375 365 403
256 461 475 473 322 123 48 78 290 106 196 95 451 344 255 482 13 173 440 429 413 263 468 409 416 0 409 224 260 435 205 487 320 459 446 307 167 201 368 193 288 33 57 475 448 22 284 223 332 377
84 449 224 483 19 277 221 112 183 357 491 426 325 487 178 470 144 35 178 169 69 492 197 284 329 228 0 427 372 198 113 204 28 409 46 9 186 239 238 444 314 2 351 129 112 151 150 193 455 370
182 466 268 214 294 135 473 473 331 402 443 178 378 271 203 365 382 91 324 238 24 180 96 170 328 43 311 0 183 59 427 488 171 29 152 259 214 361 284 287 243 111 473 129 435 342 219 4 368 366

439 382 258 251 115 99 459 162 64 257 308 279 490 436 366 76 87 387 334 3
61 331 49 429 493 434 341 267 236 0 311 100 338 393 483 329 353 370 245 2
95 109 253 148 433 36 209 265 498 244 150 16
342 190 101 313 149 24 352 475 134 392 201 355 491 198 420 281 79 432 45
341 488 400 26 484 39 493 194 253 12 0 61 335 341 498 286 30 41 306 292 3
93 211 50 79 480 472 278 74 194 121 498
327 277 291 83 79 160 419 490 160 450 425 73 381 9 468 209 478 4 371 108
197 75 223 112 20 262 57 391 164 184 0 217 433 453 242 455 314 363 397 46
1 116 405 100 137 181 199 33 388 85 241
18 7 171 242 383 250 24 259 106 122 96 297 417 179 179 80 59 78 251 8 230
82 496 179 177 254 400 285 66 94 109 0 173 244 430 15 169 56 192 474 423
249 152 487 145 447 78 18 130 417
375 292 470 413 147 194 92 316 450 358 141 53 237 52 488 227 163 456 184
395 181 98 66 66 14 262 79 79 379 141 112 448 0 446 171 476 490 251 150 3
34 366 215 283 8 433 397 368 23 383 311
142 232 188 206 480 322 39 352 448 209 147 277 260 190 423 167 487 456 29
115 361 254 278 349 4 362 432 83 456 198 107 253 322 0 297 282 22 456 44
8 125 319 136 377 275 360 499 75 254 423 136
144 389 154 233 248 181 427 179 451 302 462 200 356 364 217 74 62 246 474
275 51 354 182 288 200 111 144 466 173 30 482 113 477 382 0 248 391 172
306 373 33 490 321 166 432 159 294 207 79 449
207 172 167 397 198 21 195 30 289 110 485 470 479 118 16 127 185 169 407
429 98 119 391 200 286 487 200 421 211 272 314 416 86 319 81 0 332 268 38
8 445 187 8 361 328 75 432 153 272 269 194
386 338 312 105 178 407 269 23 414 1 43 38 39 389 356 290 148 182 94 85 4
88 262 494 218 2 483 448 166 254 105 85 96 26 222 465 282 0 373 107 157 3
44 94 81 81 369 412 214 469 252 217
80 269 41 32 434 280 164 260 154 437 96 366 375 221 336 181 477 456 226 7
2 309 60 157 103 333 406 441 376 63 386 463 81 337 298 203 9 81 0 341 77
59 494 241 47 475 274 98 381 336 73
401 208 456 167 142 89 482 169 316 397 226 10 13 137 456 263 44 243 22 42
3 13 249 19 369 218 215 151 291 336 260 170 396 304 141 480 200 106 292 0
162 182 153 254 34 30 488 43 254 84 421
315 219 245 64 230 153 365 270 471 6 48 95 488 327 277 324 41 180 491 89
211 272 446 222 471 184 90 456 479 280 7 263 136 488 197 34 89 436 280 0
494 291 463 466 2 106 308 68 170 135
172 458 499 46 98 219 339 345 373 64 29 265 302 224 491 105 102 228 198 1
93 272 364 302 364 222 66 422 446 111 496 242 23 313 152 16 56 394 239 28
0 383 0 109 155 323 208 246 339 145 291 340
155 105 124 226 79 225 482 331 234 224 95 131 347 488 446 306 117 251 490
339 464 136 198 210 131 225 409 238 475 421 373 294 356 235 62 57 107 18
5 76 383 120 0 242 433 185 280 280 284 168 337

```
126 119 238 29 120 78 238 92 57 296 61 402 294 433 137 81 376 408 185 75
220 291 477 42 352 330 291 475 73 92 190 288 491 240 394 54 64 182 404 6
177 480 0 196 140 469 499 84 140 16
122 494 327 223 339 329 82 400 479 392 24 444 335 244 350 203 208 3 142 1
88 347 392 138 414 401 317 191 163 436 127 411 378 383 261 190 206 375 16
4 223 196 67 361 39 0 89 312 246 468 426 368
190 43 64 48 3 118 100 24 227 204 49 52 71 137 459 468 457 406 32 463 320
 476 57 32 496 45 92 304 389 416 451 320 273 292 384 134 276 143 70 301 4
55 79 86 258 0 334 394 491 104 251
234 223 272 112 491 339 142 259 48 353 159 480 303 184 491 3 69 423 396 1
36 9 362 243 195 200 189 179 43 358 442 348 470 346 381 414 465 211 62 38
6 74 5 463 204 103 70 0 155 30 53 75
150 231 245 345 50 119 66 364 53 274 471 232 248 12 370 399 499 104 353 1
80 54 44 23 89 64 335 351 23 241 412 493 152 81 478 117 377 179 221 116 3
49 299 180 136 358 384 163 0 403 263 421
271 23 274 342 187 389 418 218 393 199 268 250 66 390 433 120 82 4 131 29
 132 90 153 131 173 365 408 296 429 182 491 111 178 72 237 460 144 341 13
4 38 394 131 275 494 283 204 462 0 291 163
308 498 152 424 391 218 141 204 67 384 162 160 246 387 152 266 102 341 21
0 498 284 251 59 422 176 82 288 272 288 500 5 480 139 201 31 462 119 206
41 329 460 123 399 261 99 436 482 115 0 68
494 97 162 247 39 171 39 477 56 418 372 261 253 259 434 256 66 227 402 35
 260 394 49 274 293 194 434 138 104 191 212 195 169 128 499 490 249 380 1
48 150 432 170 376 99 450 437 100 124 444 0
```

0-1 背包问题随机数据源代码

```cpp
#include <bits/stdc++.h>
#pragma GCC optimize(2)
#pragma G++ optimize(2)
#define endl "\n"
using namespace std;

void solve() {
    srand(time(NULL));
    int n = 50;
    cout << n << ' ' << rand()%200 << endl;
    for (int i = 1; i <= n; ++i)
        cout << 1 + rand() % 50 << ' ' << 1 + rand() % 200 << endl;
}

signed main() {
    ios::sync_with_stdio(false), cin.tie(0);
    freopen("IO\\in.txt", "r", stdin);
```

```
    freopen("IO\\out.txt", "w", stdout);
    solve();
    return 0;
}
```

0-1 背包问题 5 物品数据
5 41
18 135
1 170
25 79
9 163
15 106

0-1 背包问题 10 物品数据
10 123
37 132
41 3
12 85
46 193
18 75
37 102
31 4
13 41
43 152
40 90

0-1 背包问题 20 物品数据
20 111
32 144
41 185
12 17
14 54
10 81
26 28
48 169
39 63
4 45
46 196
25 76
17 109
26 50
36 9

29 82
41 2
26 72
20 123
19 166
47 178

0-1 背包问题 50 物品数据
50 186
19 36
41 44
41 67
22 173
27 79
39 134
48 69
50 52
1 184
25 9
7 169
13 98
34 26
11 71
15 168
16 104
41 48
27 70
24 179
10 103
24 132
6 150
13 66
9 195
42 9
22 175
30 54
44 18
22 58
11 32
15 150
50 2
22 105
24 185

```
46 190
27 32
19 177
50 3
34 139
39 130
23 100
49 194
11 105
30 174
25 46
49 178
25 93
32 169
26 144
30 31
```

*sudoku*随机数据源代码

```
#include <cstdlib>
#include <ctime>
#include <iostream>
using namespace std;

int sudo[9][9], hole[9][9];

bool set(int x, int y, int val) {
    if (sudo[y][x] != 0)  //非空
        return false;
    int x0, y0;
    for (x0 = 0; x0 < 9; x0++) {
        if (sudo[y][x0] == val)  //行冲突
            return false;
    }
    for (y0 = 0; y0 < 9; y0++) {
        if (sudo[y0][x] == val)  //列冲突
            return false;
    }
    for (y0 = y / 3 * 3; y0 < y / 3 * 3 + 3; y0++) {
        for (x0 = x / 3 * 3; x0 < x / 3 * 3 + 3; x0++) {
            if (sudo[y0][x0] == val)  //格冲突
                return false;
        }
    }
```

```
    sudo[y][x] = val;
    return true;
}

void reset(int x, int y) {
    sudo[y][x] = 0;
}

void initXOrd(int* xOrd)  //0~9 随机序列
{
    int i, k, tmp;
    for (i = 0; i < 9; i++) {
        xOrd[i] = i;
    }
    for (i = 0; i < 9; i++) {
        k = rand() % 9;
        tmp = xOrd[k];
        xOrd[k] = xOrd[i];
        xOrd[i] = tmp;
    }
}

bool fillFrom(int y, int val) {
    int xOrd[9];
    initXOrd(xOrd);  //生成当前行的扫描序列
    for (int i = 0; i < 9; i++) {
        int x = xOrd[i];
        if (set(x, y, val)) {
            if (y == 8)  //到了最后一行
            {
                //当前填 9 则结束, 否则从第一行填下一个数
                if (val == 9 || fillFrom(0, val + 1))
                    return true;
            } else {
                if (fillFrom(y + 1, val))  //下一行继续填当前数
                    return true;
            }
            reset(x, y);  //回溯
        }
    }
    return false;
}

void digHole(int holeCnt) {
```

```
    int idx[81];
    int i, k;
    for (i = 0; i < 81; i++) {
        hole[i / 9][i % 9] = 0;
        idx[i] = i;
    }
    for (i = 0; i < holeCnt; i++)  //随机挖洞位置
    {
        k = rand() % 81;
        int tmp = idx[k];
        idx[k] = idx[i];
        idx[i] = tmp;
    }
    for (i = 0; i < holeCnt; i++) {
        hole[idx[i] / 9][idx[i] % 9] = 1;
    }
}

void printSudo() {
    for (int y = 0; y < 9; y++) {
        for (int x = 0; x < 9; x++) {
            (hole[y][x] == 0) ? (cout << sudo[y][x] << " ") : (cout << "0
 ");
        }
        cout << endl;
    }
    cout << endl;
}

int main(int argc, char* argv[]) {
    freopen("IO\\in.txt", "r", stdin);
    freopen("IO\\out.txt", "w", stdout);
    srand((unsigned)time(NULL));
    while (!fillFrom(0, 1))
        ;
    digHole(35);
    printSudo();
    return 0;
}
```

*sudoku*测试数据 1
5 3 0 0 7 0 0 0 0
6 0 0 1 9 5 0 0 0

```
0 9 8 0 0 0 0 6 0
8 0 0 0 6 0 0 0 3
4 0 0 8 0 3 0 0 1
7 0 0 0 2 0 0 0 6
0 6 0 0 0 0 2 8 0
0 0 0 4 1 9 0 0 5
0 0 0 0 8 0 0 7 9
```

*sudoku*测试数据 2

```
7 2 0 0 0 4 5 9 0
0 0 3 9 0 0 0 0 0
0 0 0 0 0 1 0 0 3
2 6 7 4 0 0 8 3 9
4 0 9 8 3 2 6 7 0
0 8 5 7 6 0 2 1 4
0 0 2 5 9 8 0 6 7
5 7 0 1 0 0 0 2 0
0 3 0 6 2 7 0 5 1
```

*sudoku*测试数据 3

```
5 0 3 9 6 0 8 0 0
9 0 0 8 0 5 0 0 3
0 7 0 0 0 3 5 1 9
0 0 0 0 8 0 9 0 6
8 0 0 0 0 6 0 5 4
7 0 4 5 0 9 2 0 1
0 5 0 0 1 8 3 9 2
0 8 6 4 0 2 1 7 5
1 0 0 3 5 7 4 6 0
```