搜索

搜索部分简介

搜索，也就是对状态空间进行枚举，通过穷尽所有的可能来找到最优解，或者统计合法解的个数。

搜索有很多优化方式，如减小状态空间，更改搜索顺序，剪枝等。

搜索是一些高级算法的基础。在 OI 中，纯粹的搜索往往也是得到部分分的手段，但可以通过纯粹的搜索拿到满分的题目非常少。

习题

- 「kuangbin 带你飞」专题一 简单搜索
- 「kuangbin 带你飞」专题二 搜索进阶
- 洛谷搜索题单
- openjudge 搜索题单

DFS（搜索）

引入

DFS 为图论中的概念，详见 DFS（图论） 页面。在 **搜索算法** 中，该词常常指利用递归函数方便地实现暴力枚举的算法，与图论中的 DFS 算法有一定相似之处，但并不完全相同。

解释

考虑这个例子：

例题：把正整数 分解为 个不同的正整数，如 ，排在后面的数必须大于等于前面的数，输出所有方案。

对于这个问题，如果不知道搜索，应该怎么办呢？

当然是三重循环，参考代码如下：

实现：

```
for (int i = 1; i <= n; ++i)
  for (int j = i; j <= n; ++j)
    for (int k = j; k <= n; ++k)
      if (i + j + k == n) printf("%d = %d + %d + %d\n", n,
i, j, k);
```

那如果是分解成四个整数呢？再加一重循环？

那分解成小于等于 m 个整数呢？

这时候就需要用到递归搜索了。

该类搜索算法的特点在于，将要搜索的目标分成若干「层」，每层基于前几层的状态进行决策，直到达到目标状态。

考虑上述问题，即将正整数 n 分解成小于等于 m 个正整数之和，且排在后面的数必须大于等于前面的数，并输出所有方案。

设一组方案将正整数 n 分解成 k 个正整数 $a_1, a_2, \ldots, a_k$ 的和。

我们将问题分层，第 i 层决定 a_i。则为了进行第 i 层决策，我们需要记录三个状态变量：$n - \sum_{j=1}^{i} a_j$，表示后面所有正整数的和；以及 a_{i-1}，表示前一层的正整数，以确保正整数递增；以及 i，确保我们最多输出 m 个正整数。

为了记录方案，我们用 arr 数组，第 i 项表示 a_i. 注意到 arr 实际上是一个长度为 i 的栈。

代码实现：

```
int m, arr[103];  // arr 用于记录方案

void dfs(int n, int i, int a) {
  if (n == 0) {
    for (int j = 1; j <= i - 1; ++j) printf("%d ",
arr[j]);
    printf("\n");
  }
  if (i <= m) {
    for (int j = a; j <= n; ++j) {
      arr[i] = j;
      dfs(n - j, i + 1, j);  // 请仔细思考该行含义。
    }
  }
}
```

```
// 主函数
scanf("%d%d", &n, &m);
dfs(n, 1, 1);
```

例题

Luogu P1706 全排列问题

```
#include <iomanip>
#include <iostream>
using namespace std;
int n;
bool vis[50];  // 访问标记数组
int a[50];     // 排列数组，按顺序储存当前搜索结果

void dfs(int step) {
  if (step == n + 1) {  // 边界
    for (int i = 1; i <= n; i++) {
      cout << setw(5) << a[i];  // 保留5个场宽
    }
    cout << endl;
    return;
  }
  for (int i = 1; i <= n; i++) {
    if (vis[i] == 0) {  // 判断数字i是否在正在进行的全排列中
      vis[i] = 1;
      a[step] = i;
      dfs(step + 1);
      vis[i] = 0;  // 这一步不使用该数 置0后允许下一步使用
    }
  }
  return;
}

int main() {
  cin >> n;
  dfs(1);
  return 0;
}
```

BFS（搜索）

BFS 是图论中的一种遍历算法，详见 BFS。

BFS 在搜索中也很常用，将每个状态对应为图中的一个点即可。

双向搜索

本页面将简要介绍两种双向搜索算法：「双向同时搜索」和「Meet in the middle」。

双向同时搜索

定义

双向同时搜索的基本思路是从状态图上的起点和终点同时开始进行 广搜 或 深搜。

如果发现搜索的两端相遇了，那么可以认为是获得了可行解。

过程

双向广搜的步骤：

```
将开始结点和目标结点加入队列 q
标记开始结点为 1
标记目标结点为 2
while (队列 q 不为空)
{
  从 q.front() 扩展出新的 s 个结点

  如果 新扩展出的结点已经被其他数字标记过
    那么 表示搜索的两端碰撞
    那么 循环结束

  如果 新的 s 个结点是从开始结点扩展来的
    那么 将这个 s 个结点标记为 1 并且入队 q

  如果 新的 s 个结点是从目标结点扩展来的
    那么 将这个 s 个结点标记为 2 并且入队 q
```

```
}
```

Meet in the middle

Warning：本节要介绍的不是 **二分搜索**（二分搜索的另外一个译名为「折半搜索」）。

引入

Meet in the middle 算法没有正式译名，常见的翻译为「折半搜索」、「双向搜索」或「中途相遇」。

它适用于输入数据较小，但还没小到能直接使用暴力搜索的情况。

过程

Meet in the middle 算法的主要思想是将整个搜索过程分成两半，分别搜索，最后将两半的结果合并。

性质

暴力搜索的复杂度往往是指数级的，而改用 meet in the middle 算法后复杂度的指数可以减半，即让复杂度从 降到 。

例题

例题 「USACO09NOV」灯 Lights

解题思路

如果这道题暴力 DFS 找开关灯的状态，时间复杂度就是 $O(2^n)$，显然超时。不过，如果我们用 meet in middle 的话，时间复杂度可以优化至 $O(n2^{n/2})$。meet in middle 就是让我们先找一半的状态，也就是找出只使用编号为 1 到 mid 的开关能够到达的状态，再找出只使用另一半开关能到达的状态。如果前半段和后半段开启的灯互补，将这两段合并起来就得到了一种将所有灯打开的方案。具体实现时，可以把前半段的状态以及达到每种状态的最少按开关次数存储在 map 里面，搜索后半段时，每搜出一种方案，就把它与互补的第一段方案合并来更新答案。

参考代码：

```
#include <algorithm>
#include <cstdio>
#include <iostream>
#include <map>
```

```cpp
using namespace std;

int n, m, ans = 0x7fffffff;
map<long long, int> f;
long long a[40];

int main() {
  cin >> n >> m;
  a[0] = 1;
  for (int i = 1; i < n; ++i) a[i] = a[i - 1] * 2;  // 进行预处理

  for (int i = 1; i <= m; ++i) {  // 对输入的边的情况进行处理
    int u, v;
    cin >> u >> v;
    --u;
    --v;
    a[u] |= ((long long)1 << v);
    a[v] |= ((long long)1 << u);
  }

  for (int i = 0; i < (1 << (n / 2)); ++i) {  // 对前一半进行搜索
    long long t = 0;
    int cnt = 0;
    for (int j = 0; j < n / 2; ++j) {
      if ((i >> j) & 1) {
        t ^= a[j];
        ++cnt;
      }
    }
    if (!f.count(t))
      f[t] = cnt;
    else
      f[t] = min(f[t], cnt);
  }

  for (int i = 0; i < (1 << (n - n / 2)); ++i) {  // 对后一半进行搜索
    long long t = 0;
```

```
    int cnt = 0;
    for (int j = 0; j < (n - n / 2); ++j) {
      if ((i >> j) & 1) {
        t ^= a[n / 2 + j];
        ++cnt;
      }
    }
    if (f.count((((long long)1 << n) - 1) ^ t))
      ans = min(ans, cnt + f[(((long long)1 << n) - 1) ^
t]);
  }

  cout << ans;

  return 0;
}
```

外部链接

- What is meet in the middle algorithm w.r.t. competitive programming? - Quora
- [Meet in the Middle Algorithm - YouTubehttps://github.com/OI-wiki/OI-wiki)

启发式搜索

本页面将简要介绍启发式搜索及其用法。

定义

启发式搜索（英文：heuristic search）是一种在普通搜索算法的基础上引入了启发式函数的搜索算法。

启发式函数的作用是基于已有的信息对搜索的每一个分支选择都做估价，进而选择分支。简单来说，启发式搜索就是对取和不取都做分析，从中选取更优解或删去无效解。

例题

由于概念过于抽象，这里使用例题讲解。

「NOIP2005 普及组」采药

题目大意：有 N 种物品和一个容量为 W 的背包，每种物品有重量 w_i 和价值 v_i 两种属性，要求选若干个物品（每种物品只能选一次）放入背包，使背包中物品的总价值最大，且背包中物品的总重量不超过背包的容量。

解题思路

我们写一个估价函数 f，可以剪掉所有无效的 0 枝条（就是剪去大量无用不选枝条）。

估价函数 f 的运行过程如下：

我们在取的时候判断一下是不是超过了规定体积（可行性剪枝）；在不取的时候判断一下不取这个时，剩下的药所有的价值 + 现有的价值是否大于目前找到的最优解（最优性剪枝）。

示例代码：

```cpp
#include <algorithm>
#include <cstdio>
using namespace std;
const int N = 105;
int n, m, ans;

struct Node {
  int a, b;  // a 代表时间，b 代表价值
  double f;
} node[N];

bool operator<(Node p, Node q) { return p.f > q.f; }

int f(int t, int v) {  // 计算在当前时间下，剩余物品的最大价值
  int tot = 0;
  for (int i = 1; t + i <= n; i++)
    if (v >= node[t + i].a) {
      v -= node[t + i].a;
      tot += node[t + i].b;
    } else
      return (int)(tot + v * node[t + i].f);
  return tot;
}
```

```
void work(int t, int p, int v) {
  ans = max(ans, v);
  if (t > n) return;                              // 边界条件：只有n种物品
  if (f(t, p) + v > ans) work(t + 1, p, v);  // 最优性剪枝
  if (node[t].a <= p) work(t + 1, p - node[t].a, v + node[t].b);  // 可行性剪枝
}

int main() {
  scanf("%d %d", &m, &n);
  for (int i = 1; i <= n; i++) {
    scanf("%d %d", &node[i].a, &node[i].b);
    node[i].f = 1.0 * node[i].b / node[i].a;  // f为性价比
  }
  sort(node + 1, node + n + 1);  // 根据性价比排序
  work(1, m, 0);
  printf("%d\n", ans);
  return 0;
}
```

A*

本页面将简要介绍 A * 算法。

定义

A * 搜索算法（英文：A* search algorithm，A * 读作 A-star），简称 A * 算法，是一种在图形平面上，对于有多个节点的路径求出最低通过成本的算法。它属于图遍历（英文：Graph traversal）和最佳优先搜索算法（英文：Best-first search），亦是 BFS 的改进。

过程

定义起点 s，终点 t，从起点（初始状态）开始的距离函数 $g(x)$，到终点（最终状态）的距离函数 $h(x)$，$h^*(x)$[1]，以及每个点的估价函数 $f(x)=g(x)+h(x)$。

A * 算法每次从优先队列中取出一个 f 最小的元素，然后更新相邻的状态。

如果 $h \leq h*$，则 A * 算法能找到最优解。

上述条件下，如果 h 满足三角形不等式，则 A * 算法不会将重复结点加入队列。

当 $h=0$ 时，A * 算法变为 Dijkstra；当 $h=0$ 并且边权为 1 时变为 BFS。

例题

八数码

题目大意：在 3×3 的棋盘上，摆有八个棋子，每个棋子上标有 1 至 8 的某一数字。棋盘中留有一个空格，空格用 0 来表示。空格周围的棋子可以移到空格中，这样原来的位置就会变成空格。给出一种初始布局和目标布局（为了使题目简单，设目标状态如下），找到一种从初始布局到目标布局最少步骤的移动方法。

```
1  123
2  804
3  765
```

解题思路

h 函数可以定义为，不在应该在的位置的数字个数。

容易发现 h 满足以上两个性质，此题可以使用 A * 算法求解。

参考代码：

```cpp
#include <algorithm>
#include <cstdio>
#include <cstring>
#include <queue>
#include <set>
using namespace std;
const int dx[4] = {1, -1, 0, 0}, dy[4] = {0, 0, 1, -1};
int fx, fy;
char ch;

struct matrix {
  int a[5][5];

  bool operator<(matrix x) const {
    for (int i = 1; i <= 3; i++)
```

```
      for (int j = 1; j <= 3; j++)
        if (a[i][j] != x.a[i][j]) return a[i][j] < x.a[i]
[j];
    return false;
  }
} f, st;

int h(matrix a) {
  int ret = 0;
  for (int i = 1; i <= 3; i++)
    for (int j = 1; j <= 3; j++)
      if (a.a[i][j] != st.a[i][j]) ret++;
  return ret;
}

struct node {
  matrix a;
  int t;

  bool operator<(node x) const { return t + h(a) > x.t +
h(x.a); }
} x;

priority_queue<node> q;  // 搜索队列
set<matrix> s;           // 防止搜索队列重复

int main() {
  st.a[1][1] = 1;  // 定义标准表
  st.a[1][2] = 2;
  st.a[1][3] = 3;
  st.a[2][1] = 8;
  st.a[2][2] = 0;
  st.a[2][3] = 4;
  st.a[3][1] = 7;
  st.a[3][2] = 6;
  st.a[3][3] = 5;
  for (int i = 1; i <= 3; i++)  // 输入
    for (int j = 1; j <= 3; j++) {
      scanf(" %c", &ch);
      f.a[i][j] = ch - '0';
```

```
    }
  q.push({f, 0});
  while (!q.empty()) {
    x = q.top();
    q.pop();
    if (!h(x.a)) {  // 判断是否与标准矩阵一致
      printf("%d\n", x.t);
      return 0;
    }
    for (int i = 1; i <= 3; i++)
      for (int j = 1; j <= 3; j++)
        if (!x.a.a[i][j]) fx = i, fy = j;  // 查找空格子（0号点）的位置
    for (int i = 0; i < 4; i++) {  // 对四种移动方式分别进行搜索
      int xx = fx + dx[i], yy = fy + dy[i];
      if (1 <= xx && xx <= 3 && 1 <= yy && yy <= 3) {
        swap(x.a.a[fx][fy], x.a.a[xx][yy]);
        if (!s.count(x.a))
          s.insert(x.a),
              q.push({x.a, x.t + 1});  // 这样移动后，将新的情况放入搜索队列中
        swap(x.a.a[fx][fy], x.a.a[xx][yy]);  // 如果不这样移动的情况
      }
    }
  }
  return 0;
}
```

注：对于 k 短路问题，原题已经可以构造出数据使得 A* 算法无法通过，故本题思路仅供参考，A* 算法非正解，正解为可持久化可并堆做法，请移步 k 短路问题

k 短路

按顺序求一个有向图上从结点 s 到结点 t 的所有路径最小的前任意多（不妨设为 k）个。

解题思路

很容易发现，这个问题很容易转化成用 A * 算法解决问题的标准程式。

初始状态为处于结点 s，最终状态为处于结点 t，距离函数为从 s 到当前结点已经走过的距离，估价函数为从当前结点到结点 t 至少要走过的距离，也就是当前结点到结点 t 的最短路。

就这样，我们在预处理的时候反向建图，计算出结点 t 到所有点的最短路，然后将初始状态塞入优先队列，每次取出 $f(x)=g(x)+h(x)$ 最小的一项，计算出其所连结点的信息并将其也塞入队列。当你第 k 次走到结点 t 时，也就算出了结点 s 到结点 t 的 k 短路。

由于设计的距离函数和估价函数，每个状态需要存储两个参数，当前结点 x 和已经走过的距离 v。

我们可以在此基础上加一点小优化：由于只需要求出第 k 短路，所以当我们第 $k+1$ 次或以上走到该结点时，直接跳过该状态。因为前面的 k 次走到这个点的时候肯定能因此构造出 k 条路径，所以之后再加边更无必要。

参考代码：

```
#include <algorithm>
#include <cstdio>
#include <cstring>
#include <queue>
using namespace std;
const int maxn = 5010;
const int maxm = 400010;
const double inf = 2e9;
int n, m, k, u, v, cur, h[maxn], nxt[maxm], p[maxm],
cnt[maxn], ans;
int cur1, h1[maxn], nxt1[maxm], p1[maxm];
double e, ww, w[maxm], f[maxn];
double w1[maxm];
bool tf[maxn];

void add_edge(int x, int y, double z) {  // 正向建图函数
  cur++;
  nxt[cur] = h[x];
  h[x] = cur;
  p[cur] = y;
  w[cur] = z;
}

void add_edge1(int x, int y, double z) {  // 反向建图函数
```

```
  cur1++;
  nxt1[cur1] = h1[x];
  h1[x] = cur1;
  p1[cur1] = y;
  w1[cur1] = z;
}

struct node {  // 使用A*时所需的结构体
  int x;
  double v;

  bool operator<(node a) const { return v + f[x] > a.v +
f[a.x]; }
};

priority_queue<node> q;

struct node2 {  // 计算t到所有结点最短路时所需的结构体
  int x;
  double v;

  bool operator<(node2 a) const { return v > a.v; }
} x;

priority_queue<node2> Q;

int main() {
  scanf("%d%d%lf", &n, &m, &e);
  while (m--) {
    scanf("%d%d%lf", &u, &v, &ww);
    add_edge(u, v, ww);   // 正向建图
    add_edge1(v, u, ww);  // 反向建图
  }
  for (int i = 1; i < n; i++) f[i] = inf;
  Q.push({n, 0});
  while (!Q.empty()) {  // 计算t到所有结点的最短路
    x = Q.top();
    Q.pop();
    if (tf[x.x]) continue;
    tf[x.x] = true;
```

```
    f[x.x] = x.v;
    for (int j = h1[x.x]; j; j = nxt1[j]) Q.push({p1[j],
x.v + w1[j]});
  }
  k = (int)e / f[1];
  q.push({1, 0});
  while (!q.empty()) {  // 使用A*算法
    node x = q.top();
    q.pop();
    cnt[x.x]++;
    if (x.x == n) {
      e -= x.v;
      if (e < 0) {
        printf("%d\n", ans);
        return 0;
      }
      ans++;
    }
    for (int j = h[x.x]; j; j = nxt[j])
      if (cnt[p[j]] <= k && x.v + w[j] <= e) q.push({p[j],
x.v + w[j]});
  }
  printf("%d\n", ans);
  return 0;
}
```

参考资料与注释

1. 此处的 h 意为 heuristic。详见 启发式搜索 - 维基百科 和 A*search algorithm - Wikipedia 的 Bounded relaxation 一节。 ↩

迭代加深搜索

定义

迭代加深是一种 **每次限制搜索深度的** 深度优先搜索。

解释

迭代加深搜索的本质还是深度优先搜索，只不过在搜索的同时带上了一个深度 d，当 d 达到设定的深度时就返回，一般用于找最优解。如果一次搜索没有找到合法的解，就让设定的深度加一，重新从根开始。

既然是为了找最优解，为什么不用 BFS 呢？我们知道 BFS 的基础是一个队列，队列的空间复杂度很大，当状态比较多或者单个状态比较大时，使用队列的 BFS 就显出了劣势。事实上，迭代加深就类似于用 DFS 方式实现的 BFS，它的空间复杂度相对较小。

当搜索树的分支比较多时，每增加一层的搜索复杂度会出现指数级爆炸式增长，这时前面重复进行的部分所带来的复杂度几乎可以忽略，这也就是为什么迭代加深是可以近似看成 BFS 的。

过程

首先设定一个较小的深度作为全局变量，进行 DFS。每进入一次 DFS，将当前深度加一，当发现 d 大于设定的深度 $limit$ 就返回。如果在搜索的途中发现了答案就可以回溯，同时在回溯的过程中可以记录路径。如果没有发现答案，就返回到函数入口，增加设定深度，继续搜索。

实现（伪代码）：

```
IDDFS(u,d)
    if d>limit
        return
    else
        for each edge (u,v)
            IDDFS(v,d+1)
return
```

注意事项

在大多数的题目中，广度优先搜索还是比较方便的，而且容易判重。当发现广度优先搜索在空间上不够优秀，而且要找最优解的问题时，就应该考虑迭代加深。

IDA*

前置知识：A* 算法、迭代加深搜索。

本页面将简要介绍 IDA * 算法。

定义

IDA * 为采用了迭代加深算法的 A * 算法。

优点

由于 IDA * 改成了深度优先的方式，相对于 A * 算法，它的优点如下：

1. 不需要判重，不需要排序，利于深度剪枝。
2. 空间需求减少：每个深度下实际上是一个深度优先搜索，不过深度有限制，使用 DFS 可以减小空间消耗。

缺点

1. 重复搜索：即使前后两次搜索相差微小，回溯过程中每次深度变大都要再次从头搜索。

实现（伪代码）

```
Procedure IDA_STAR(StartState)
Begin
  PathLimit := H(StartState) - 1;
  Succes := False;
  Repeat
    inc(PathLimit);
    StartState.g = 0;
    Push(OpenStack, StartState);
    Repeat
      CurrentState := Pop(OpenStack);
      If Solution(CurrentState) then
        Success = True
      Elseif PathLimit >= CurrentState.g + H(CurrentState)
then
        For each Child(CurrentState) do
          Push(OpenStack, Child(CurrentState));
    until Success or empty(OpenStack);
```

```
  until Success or ResourceLimtsReached;
end;
```

例题

埃及分数

在古埃及，人们使用单位分数的和（即 $\frac{1}{a}$，$a \in \mathbb{N}^*$）表示一切有理数。例如，$\frac{2}{3} = \frac{1}{2} + \frac{1}{6}$，但不允许 $\frac{2}{3} = \frac{1}{3} + \frac{1}{3}$，因为在加数中不允许有相同的。

对于一个分数 $\frac{a}{b}$，表示方法有很多种，其中加数少的比加数多的好，如果加数个数相同，则最小的分数越大越好。例如，$\frac{19}{45} = \frac{1}{5} + \frac{1}{6} + \frac{1}{18}$ 是最优方案。

输入整数 a, b $(0 < a < b < 500)$，试编程计算最佳表达式。

样例输入：

```
495 499
```

样例输出：

```
Case 1: 495/499=1/2+1/5+1/6+1/8+1/3992+1/14970
```

解题思路

这道题目理论上可以用回溯法求解，但是解答树会非常「恐怖」——不仅深度没有明显的上界，而且加数的选择理论上也是无限的。换句话说，如果用宽度优先遍历，连一层都扩展不完，因为每一层都是 **无限大** 的。

解决方案是采用迭代加深搜索：从小到大枚举深度上限 $maxd$，每次执行只考虑深度不超过 $maxd$ 的节点。这样，只要解的深度有限，则一定可以在有限时间内枚举到。

深度上限 $maxd$ 还可以用来 **剪枝**。按照分母递增的顺序来进行扩展，如果扩展到 i 层时，前 i 个分数之和为 $\frac{c}{d}$，而第 i 个分数为 $\frac{1}{e}$，则接下来至少还需要 $\frac{\frac{a}{b}-\frac{c}{d}}{\frac{1}{e}}$ 个分数，总和才能达到 $\frac{a}{b}$。例如，当前搜索到 $\frac{19}{45} = \frac{1}{5} + \frac{1}{100} + \cdots$，则后面的分数每个最大为 $\frac{1}{101}$，至少需要 $\frac{\frac{19}{45}-\frac{1}{5}}{\frac{1}{101}} = 23$ 项总和才能达到 $\frac{19}{45}$，因此前 22 次迭代是根本不会考虑这棵子树的。这里的关键在于：可以估计至少还要多少步才能出解。

注意，这里使用 **至少** 一词表示估计都是乐观的。形式化地，设深度上限为 $maxd$，当前结点 n 的深度为 $g(n)$，乐观估价函数为 $h(n)$，则当 $g(n) + h(n) > maxd$ 时应该剪枝。这样的算法就是 IDA*。当然，在实战中不需要严格地在代码里写出 $g(n)$ 和 $h(n)$，只需要像刚才那样设计出乐观估价函数，想清楚在什么情况下不可能在当前的深度限制下出解即可。

> 如果可以设计出一个乐观估价函数，预测从当前结点至少还需要扩展几层结点才有可能得到解，则迭代加深搜索变成了 IDA * 算法。

示例代码

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

using namespace std;

const int MAX_E = 1e7;
int a, b;
```

```cpp
vector<int> ans;
vector<int> current;

inline bool better() { return ans.empty() ||
current.back() < ans.back(); }

bool dfs(int d, long a, long b, int e) {
  if (d == 0) {
    if (a == 0 && better()) ans = current;
    return a == 0;
  }

  long _gcd = gcd(a, b);
  a /= _gcd;
  b /= _gcd;

  bool solved = false;
  // the min value of e:
  // a/b - 1/e >= 0
  // e >= b/a
  int e1 = max(e + 1, int((b + a - 1) / a));
  // b/a <= e <= MAX_E
  // b/a <= MAX_E
  if (b > a * MAX_E) {
    return false;
  }
  for (;; e1++) {
    // the max value of e:
    // d * (1/e) >= a/b
    // d/e >= a/b
    if (d * b < a * e1) {
      return solved;
    }
    current.push_back(e1);
    // a/b - 1/e
    solved |= dfs(d - 1, a * e1 - b, b * e1, e1);
    current.pop_back();
  }
  return solved;
}
```

```
int solve() {
  ans.clear();
  current.clear();
  for (int maxd = 1; maxd <= 100; maxd++)
    if (dfs(maxd, a, b, 1)) return maxd;
  return -1;
}

int main() {
  int kase = 0;
  while (cin >> a >> b) {
    int maxd = solve();
    cout << "Case " << ++kase << ": ";
    if (maxd > 0) {
      cout << a << "/" << b << "=";
      for (int i = 0; i < maxd - 1; i++) cout << "1/" <<
ans[i] << "+";
      cout << "1/" << ans[maxd - 1] << "\n";
    } else
      cout << "No solution.\n";
  }
  return 0;
}
```

习题

- UVa1343 旋转游戏

回溯法

本页面将简要介绍回溯法的概念和应用。

简介

回溯法是一种经常被用在 深度优先搜索（DFS） 和 广度优先搜索（BFS） 的技巧。

其本质是：走不通就回头。

过程

1. 构造空间树；
2. 进行遍历；
3. 如遇到边界条件，即不再向下搜索，转而搜索另一条链；
4. 达到目标条件，输出结果。

例题

USACO 1.5.4 Checker Challenge

现在有一个如下的 6×6 的跳棋棋盘，有六个棋子被放置在棋盘上，使得每行，每列，每条对角线（包括两条主对角线的所有对角线）上都至多有一个棋子。

```
  0   1   2   3   4   5   6
    ------------------------
1 |   | O |   |   |   |   |
    ------------------------
2 |   |   |   | O |   |   |
    ------------------------
3 |   |   |   |   |   | O |
    ------------------------
4 | O |   |   |   |   |   |
    ------------------------
5 |   |   | O |   |   |   |
    ------------------------
6 |   |   |   |   | O |   |
    ------------------------
```

上面的布局可以用序列 $\{2,4,6,1,3,5\}$ 来描述，第 i 个数字表示在第 i 行的第 a_i 列有一个棋子，如下所示

行号 i：$\{1,2,3,4,5,6\}$

列号 a_i：$\{2,4,6,1,3,5\}$

这只是跳棋放置的一个方案。请编一个程序找出所有方案并把它们以上面的序列化方法输出，按字典顺序排列。你只需输出前 3 个解并在最后一行输出解的总个数。特别注意：你需要优化你的程序以保证在更大棋盘尺寸下的程序效率。

参考代码：

```
// 该代码为回溯法的 DFS 实现
#include <cstdio>
int ans[14], check[3][28] = {0}, sum = 0, n;

void eq(int line) {
  if (line > n) {  // 如果已经搜索完n行
    sum++;
    if (sum > 3)
      return;
```

```
    else {
      for (int i = 1; i <= n; i++) printf("%d ", ans[i]);
      printf("\n");
      return;
    }
  }
  for (int i = 1; i <= n; i++) {
    if ((!check[0][i]) && (!check[1][line + i]) &&
        (!check[2][line - i + n])) {  // 判断在某位置放置是否
合法
      ans[line] = i;
      check[0][i] = 1;
      check[1][line + i] = 1;
      check[2][line - i + n] = 1;
      eq(line + 1);
      // 向下递归后进行回溯，方便下一轮递归
      check[0][i] = 0;
      check[1][line + i] = 0;
      check[2][line - i + n] = 0;
    }
  }
}

int main() {
  scanf("%d", &n);
  eq(1);
  printf("%d", sum);
  return 0;
}
```

迷宫

现有一个尺寸为 $N \times M$ 的迷宫，迷宫里有 T 处障碍，障碍处不可通过。给定起点坐标和终点坐标，且每个方格最多经过一次，问有多少种从起点坐标到终点坐标的方案。在迷宫中移动有上、下、左、右四种移动方式，每次只能移动一个方格。数据保证起点上没有障碍。

```
// 该代码为回溯法的 BFS 实现
#include <cstdio>
#include <cstring>
#include <queue>
using namespace std;
```

```
int n, m, k, x, y, a, b, ans;
int dx[4] = {0, 0, 1, -1}, dy[4] = {1, -1, 0, 0};  // 四个
方向
bool vis[6][6];

struct oo {
  int x, y, used[6][6];
};

oo sa;

void bfs() {
  queue<oo> q;
  sa.x = x;
  sa.y = y;
  sa.used[x][y] = 1;
  q.push(sa);
  while (!q.empty()) {  // BFS队列
    oo now = q.front();
    q.pop();
    for (int i = 0; i < 4; i++) {  // 枚举向四个方向走
      int sx = now.x + dx[i];
      int sy = now.y + dy[i];
      if (now.used[sx][sy] || vis[sx][sy] || sx == 0 || sy
== 0 || sx > n ||
          sy > m)
        continue;
      if (sx == a && sy == b) {
        ans++;
        continue;
      }
      sa.x = sx;
      sa.y = sy;
      memcpy(sa.used, now.used, sizeof(now.used));
      sa.used[sx][sy] = 1;
      q.push(sa);  // 假设向此方向走，放入BFS队列
    }
  }
}
```

```
int main() {
  scanf("%d%d%d", &n, &m, &k);
  scanf("%d%d%d%d", &x, &y, &a, &b);
  for (int i = 1, aa, bb; i <= k; i++) {
    scanf("%d%d", &aa, &bb);
    vis[aa][bb] = 1;  // 障碍位置不可通过
  }
  bfs();
  printf("%d", ans);
  return 0;
}
```

Dancing Links

本页面将介绍精确覆盖问题、重复覆盖问题，解决这两个问题的算法「X 算法」，以及用来优化 X 算法的双向十字链表 Dancing Link。本页也将介绍如何在建模的配合下使用 DLX 解决一些搜索题。

精确覆盖问题

定义

精确覆盖问题（英文：Exact Cover Problem）是指给定许多集合 $S_i(1 \le i \le n)$ 以及一个集合 X，求满足以下条件的无序多元组 $(T_1, T_2, \cdots, T_m)$：

1. $\forall i, j \in [1, m], T_i \bigcap T_j = \varnothing (i \neq j)$
2. $X = \bigcup_{i=1}^{m} T_i$
3. $\forall i \in [1, m], T_i \in \{S_1, S_2, \cdots, S_n\}$

解释

例如，若给出

$$
\begin{aligned}
S_1 &= \{5, 9, 17\} \\
S_2 &= \{1, 8, 119\} \\
S_3 &= \{3, 5, 17\} \\
S_4 &= \{1, 8\} \\
S_5 &= \{3, 119\} \\
S_6 &= \{8, 9, 119\} \\
X &= \{1, 3, 5, 8, 9, 17, 119\}
\end{aligned}
$$

则 (S_1, S_4, S_5) 为一组合法解。

问题转化

将 $\bigcup_{i=1}^{n} S_i$ 中的所有数离散化，可以得到这么一个模型：

> 给定一个 01 矩阵，你可以选择一些行（row），使得最终每列（column）[1]都恰好有一个 1。 举个例子，我们对上文中的例子进行建模，可以得到这么一个矩阵：

$$
\begin{pmatrix}
0 & 0 & 1 & 0 & 1 & 1 & 0 \\
1 & 0 & 0 & 1 & 0 & 0 & 1 \\
0 & 1 & 1 & 0 & 0 & 1 & 0 \\
1 & 0 & 0 & 1 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 & 0 & 0 & 1 \\
0 & 0 & 0 & 1 & 1 & 0 & 1
\end{pmatrix}
$$

> 其中第 i 行表示着 S_i，而这一行的每个数依次表示 $[1 \in S_i], [3 \in S_i], [5 \in S_i], \cdots, [119 \in S_i]$。

实现

暴力1

一种方法是枚举选择哪些行，最后检查这个方案是否合法。

因为每一行都有选或者不选两种状态，所以枚举行的时间复杂度是 $O(2^n)$ 的；

而每次检查都需要 $O(nm)$ 的时间复杂度。所以总的复杂度是 $O(nm2^n)$ 。

实现：

```
int ok = 0;
for (int state = 0; state < 1 << n; ++state) {  // 枚举每行是否被选
```

```
  for (int i = 1; i <= n; ++i)
    if ((1 << i - 1) & state)
      for (int j = 1; j <= m; ++j) a[i][j] = 1;
  int flag = 1;
  for (int j = 1; j <= m; ++j)
    for (int i = 1, bo = 0; i <= n; ++i)
      if (a[i][j]) {
        if (bo)
          flag = 0;
        else
          bo = 1;
      }
  if (!flag)
    continue;
  else {
    ok = 1;
    for (int i = 1; i <= n; ++i)
      if ((1 << i - 1) & state) printf("%d ", i);
    puts("");
  }
  memset(a, 0, sizeof(a));
}
if (!ok) puts("No solution.");
```

暴力2

考虑到 01 矩阵的特殊性质，每一行都可以看做一个 m 位二进制数。

因此原问题转化为

> 给定 n 个 m 位二进制数，要求选择一些数，使得任意两个数的与都为 0，且所有数的或为 2^m-1。`tmp` 表示的是截至目前被选中的二进制数的或。

因为每一行都有选或者不选两种状态，所以枚举行的时间复杂度为 $O(2^n)$；

而每次计算 `tmp` 都需要 $O(n)$ 的时间复杂度。所以总的复杂度为 $O(n\cdot 2^n)$。

实现：

```
int ok = 0;
for (int i = 1; i <= n; ++i)
  for (int j = m; j >= 1; --j) num[i] = num[i] << 1 | a[i]
[j];
```

```
for (int state = 0; state < 1 << n; ++state) {
  int tmp = 0;
  for (int i = 1; i <= n; ++i)
    if ((1 << i - 1) & state) {
      if (tmp & num[i]) break;
      tmp |= num[i];
    }
  if (tmp == (1 << m) - 1) {
    ok = 1;
    for (int i = 1; i <= n; ++i)
      if ((1 << i - 1) & state) printf("%d ", i);
    puts("");
  }
}
if (!ok) puts("No solution.");
```

重复覆盖问题

重复覆盖问题与精确覆盖问题类似，但没有对元素相似性的限制。下文介绍的 X 算法 原本针对精确覆盖问题，但经过一些修改和优化（已标注在其中）同样可以高效地解决重复覆盖问题。

X算法

Donald E. Knuth 提出了 X 算法 (Algorithm X)，其思想与刚才的暴力差不多，但是方便优化。

过程

继续以上文中中提到的例子为载体，得到一个这样的 01 矩阵：

$$\begin{pmatrix} 0 & 0 & 1 & 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 1 & 0 & 1 \end{pmatrix}$$

1. 此时第一行有 3 个 1，第二行有 3 个 1，第三行有 3 个 1，第四行有 2 个 1，第五行有 2 个 1，第六行有 3 个 1。选择第一行，将它删除，并将所有 所在的列打上标记；

$$\begin{pmatrix} 0 & 0 & 1 & 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 1 & 0 & 1 \end{pmatrix}$$

2. 选择所有被标记的列，将它们删除，并将这些列中含 1 的行打上标记（重复覆盖问题无需打标记）；

$$\begin{pmatrix} 0 & 0 & 1 & 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 1 & 0 & 1 \end{pmatrix}$$

3. 选择所有被标记的行，将它们删除；

$$\begin{pmatrix} 0 & 0 & 1 & 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 1 & 0 & 1 \end{pmatrix}$$

这表示这一行已被选择，且这一行的所有 1 所在的列不能有其他 1 了。

于是得到一个新的小 01 矩阵：

$$\begin{pmatrix} 1 & 0 & 1 & 1 \\ 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \end{pmatrix}$$

4. 此时第一行（原来的第二行）有 3 个 1，第二行（原来的第四行）有 2 个 1，第三行（原来的第五行）有 2 个 1。选择第一行（原来的第二行），将它删除，并将所有 1 所在的列打上标记；

$$\begin{pmatrix} 1 & 0 & 1 & 1 \\ 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \end{pmatrix}$$

5. 选择所有被标记的列，将它们删除，并将这些列中含 的行打上标记；

$$\begin{pmatrix} 1 & 0 & 1 & 1 \\ 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \end{pmatrix}$$

6. 选择所有被标记的行，将它们删除；

$$\begin{pmatrix} 1 & 0 & 1 & 1 \\ 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \end{pmatrix}$$

这样就得到了一个空矩阵。但是上次删除的行 `1 0 1 1` 不是全 1 的，说明选择有误；

7. 回溯到步骤 4，考虑选择第二行（原来的第四行），将它删除，并将所有 1 所在的列打上标记；

$$\begin{pmatrix} 1 & 0 & 1 & 1 \\ 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \end{pmatrix}$$

8. 选择所有被标记的列，将它们删除，并将这些列中含 的行打上标记；

$$\begin{pmatrix} 1 & 0 & 1 & 1 \\ 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \end{pmatrix}$$

9. 选择所有被标记的行，将它们删除；

$$\begin{pmatrix} 1 & 0 & 1 & 1 \\ 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \end{pmatrix}$$

于是我们得到了这样的一个矩阵：

$$\begin{pmatrix} 1 & 1 \end{pmatrix}$$

10. 此时第一行（原来的第五行）有 2 个 1，将它们全部删除，得到一个空矩阵：

$$()$$

11. 上一次删除的时候，删除的是全 1 的行，因此成功，算法结束。

    答案即为被删除的三行：1，4，5。

强烈建议自己模拟一遍矩阵删除、还原与回溯的过程后，再接着阅读下文。

通过上述步骤，可将 X 算法的流程概括如下：

1. 对于现在的矩阵 M，选择并标记一行 r，将 r 添加至 S 中；
2. 如果尝试了所有的 r 却无解，则算法结束，输出无解；
3. 标记与 r 相关的行 r_i 和 c_i（相关的行和列与 X 算法 中第 2 步定义相同，下同）；
4. 删除所有标记的行和列，得到新矩阵 M'；
5. 如果 M' 为空，且 r 为全 1，则算法结束，输出被删除的行组成的集合 S；

   如果 M' 为空，且 r 不全为 1，则恢复与 r 相关的行 r_i 以及列 c_i，跳转至步骤 1；

   如果 M' 不为空，则跳转至步骤 1。

不难看出，X 算法需要大量的「删除行」、「删除列」和「恢复行」、「恢复列」的操作。

一个朴素的想法是，使用一个二维数组存放矩阵，再用四个数组分别存放每一行与之相邻的行编号，每次删除和恢复仅需更新四个数组中的元素。但由于一般问题的矩阵中 0 的数量远多于 1 的数量，这样做的空间复杂度难以接受。

Donald E. Knuth 想到了用双向十字链表来维护这些操作。

而在双向十字链表上不断跳跃的过程被形象地比喻成「跳跃」，因此被用来优化 X 算法的双向十字链表也被称为「Dancing Links」。

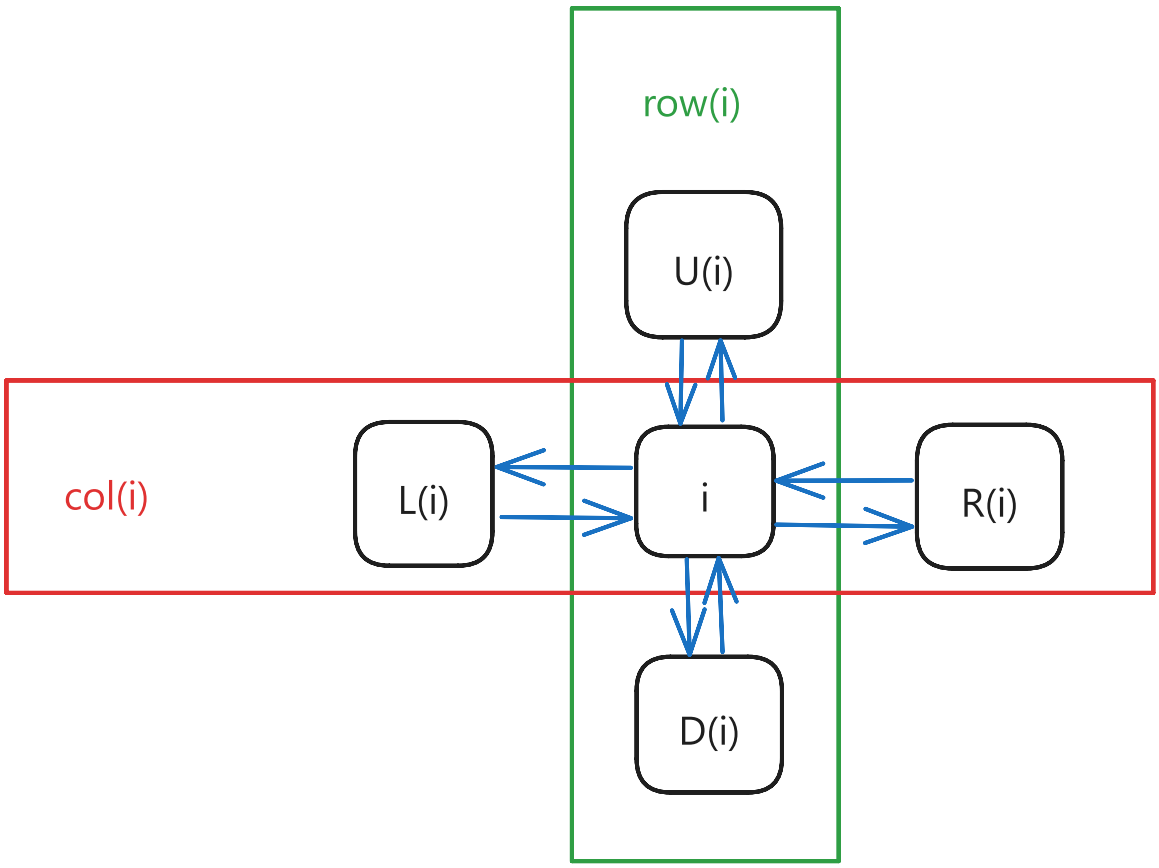
Dancing Links 优化的 X 算法

预编译命令

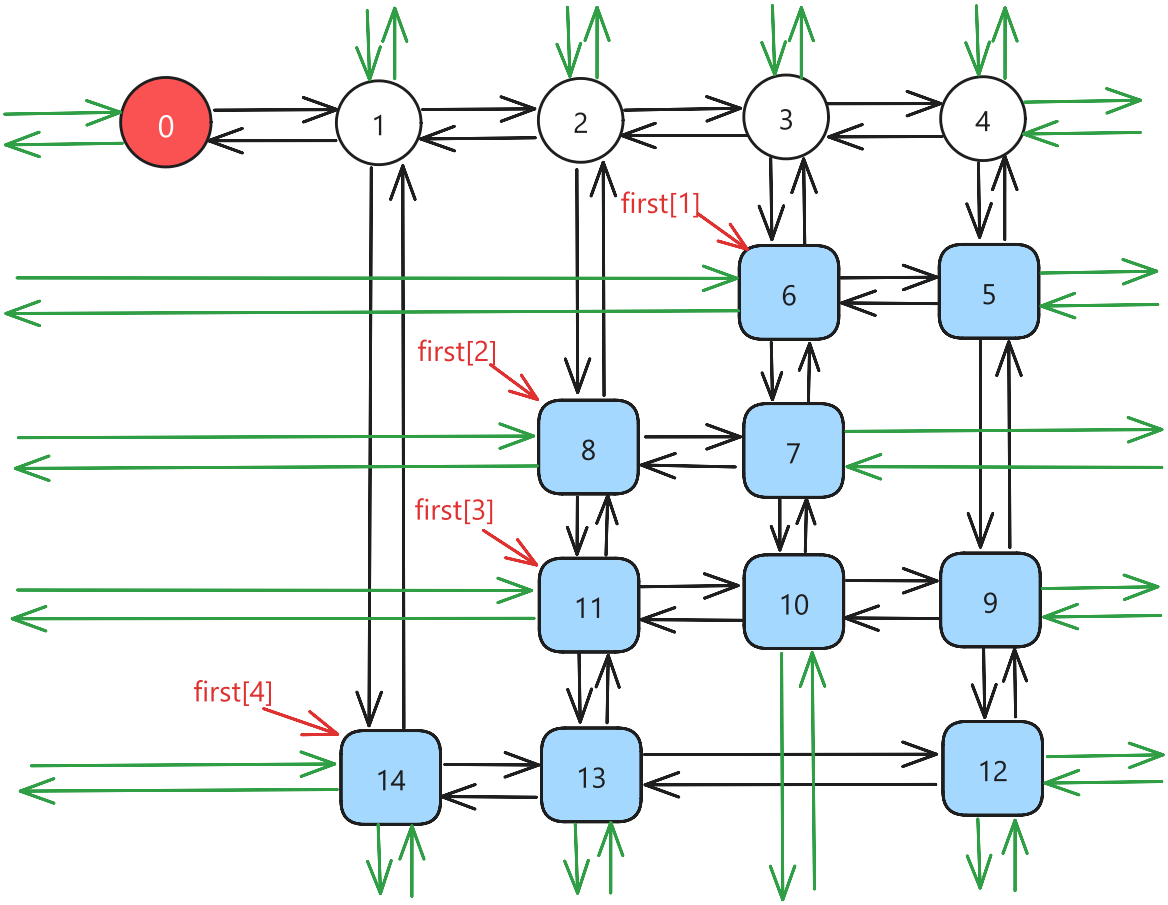
```
#define IT(i, A, x) for (i = A[x]; i != x; i = A[i])
```

定义

双向十字链表中存在四个指针域，分别指向上、下、左、右的元素；且每个元素 i 在整个双向十字链表系中都对应着一个格子，因此还要表示 i 所在的列和所在的行，如图所示：



大型的双向链表则更为复杂：



每一行都有一个行首指示，每一列都有一个列指示。

行首指示为 `first[]`，列指示是我们新建的 $c+1$ 个哨兵结点。值得注意的是，**行首指示并非是链表中的哨兵结点**。它是虚拟的，类似于邻接表中的 `first[]` 数组，**直接指向** 这一行中的首元素。

同时，每一列都有一个 `siz[]` 表示这一列的元素个数。

特殊地，0 号结点无右结点等价于这个 Dancing Links 为空。

```
constexpr int MS = 1e5 + 5;
int n, m, idx, first[MS], siz[MS];
int L[MS], R[MS], U[MS], D[MS];
int col[MS], row[MS];
```

过程

- remove操作

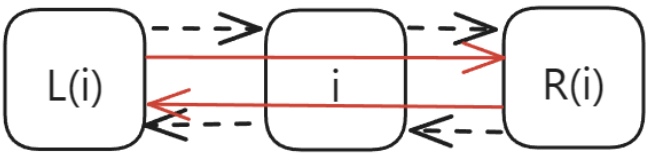
  remove 操作

  `remove(c)` 表示在 Dancing Links 中删除第 c 列以及与其相关的行和列。

  先将 c 删除，此时：

  - c 左侧的结点的右结点应为 c 的右结点。
  - c 右侧的结点的左结点应为 c 的左结点。

  即 `L[R[c]] = L[c], R[L[c]] = R[c];`。

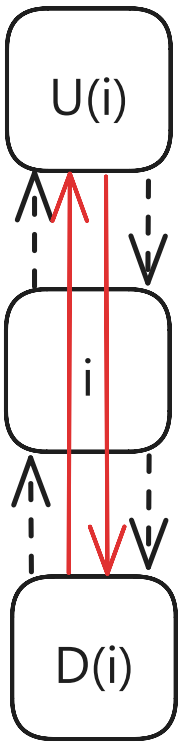
  然后顺着这一列往下走，把走过的每一行都删掉。

  如何删掉每一行呢？枚举当前行的指针 j，此时：

  - j 上方的结点的下结点应为 j 的下结点。
  - j 下方的结点的上结点应为 j 的上结点。

  注意要修改每一列的元素个数。

  即 `U[D[j]] = U[j], D[U[j]] = D[j], --siz[col[j]];`。



remove 函数的代码实现如下：

```
void remove(const int &c) {
  int i, j;
  L[R[c]] = L[c], R[L[c]] = R[c];
  // 顺着这一列从上往下遍历
  IT(i, D, c)
  // 顺着这一行从左往右遍历
  IT(j, R, i)
  U[D[j]] = U[j], D[U[j]] = D[j], --siz[col[j]];
}
```

- recover 操作

  `recover(c)` 表示在 Dancing Links 中还原第 列以及与其相关的行和列。

  `recover(c)` 即 `remove(c)` 的逆操作，这里不再赘述。

  值得注意的是，`recover(c)` 的所有操作的顺序与 `remove(c)` 的操作恰好相反。

  `recover(c)` 的代码实现如下：

```
void recover(const int &c) {
  int i, j;
  IT(i, U, c) IT(j, L, i) U[D[j]] = D[U[j]] = j,
++siz[col[j]];
  L[R[c]] = R[L[c]] = c;
}
```

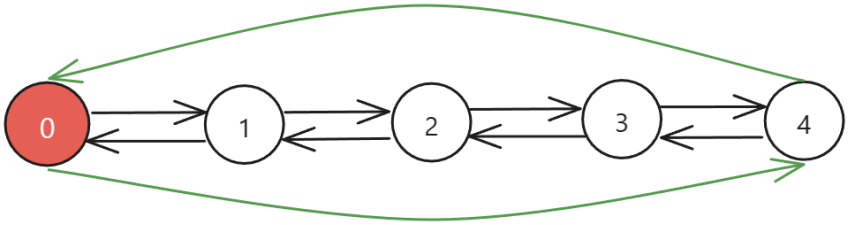
- build 操作

`build(r, c)` 表示新建一个大小为 $r \times c$，即有 r 行，c 列的 Dancing Links。

新建 $c+1$ 个结点作为列指示。

第 i 个点的左结点为 $i-1$，右结点为 $i+1$，上结点为 i，下结点为 i。特殊地，0 结点的左结点为 c，c 结点的右结点为 0。

于是我们得到了一个环状双向链表：



这样就初始化了一个 Dancing Links。

`build(r, c)` 的代码实现如下：

```
void build(const int &r, const int &c) {
  n = r, m = c;
  for (int i = 0; i <= c; ++i) {
    L[i] = i - 1, R[i] = i + 1;
    U[i] = D[i] = i;
  }
  L[0] = c, R[c] = 0, idx = c;
  memset(first, 0, sizeof(first));
  memset(siz, 0, sizeof(siz));
}
```

- insert 操作

`insert(r, c)` 表示在第 行，第 列插入一个结点。

插入操作分为两种情况：

- 如果第 r 行没有元素，那么直接插入一个元素，并使 `first[r]` 指向这个元素。

  这可以通过 `first[r] = L[idx] = R[idx] = idx;` 来实现。
- 如果第 r 行有元素，那么将这个新元素用一种特殊的方式与 c 和 $first(r)$ 连接起来。

  设这个新元素为 idx，然后：

  - 把 idx 插入到 c 的正下方，此时：
    - idx 下方的结点为原来 c 的下结点；
    - idx 下方的结点（即原来 c 的下结点）的上结点为 idx；
    - idx 的上结点为 c；
    - c 的下结点为 idx。

    注意记录 idx 的所在列和所在行，以及更新这一列的元素个数。

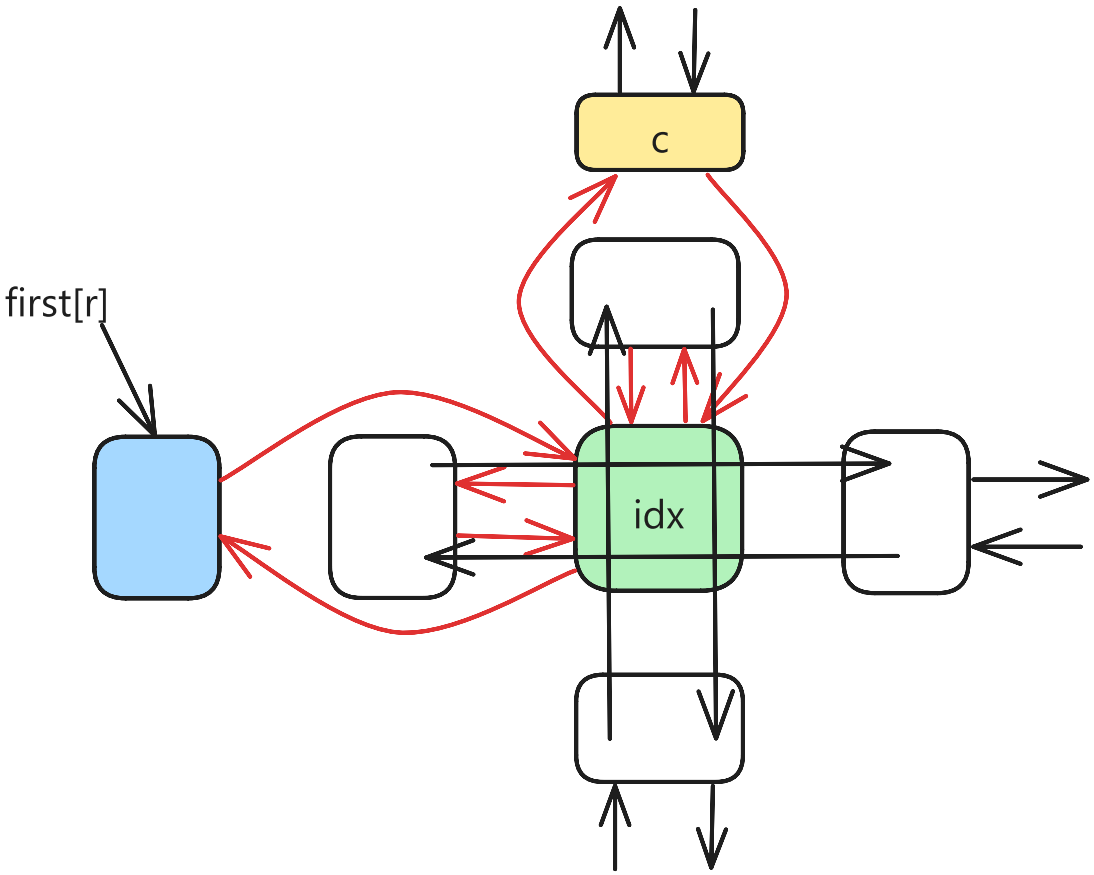
    ```
    col[++idx] = c, row[idx] = r, ++siz[c];
    U[idx] = c, D[idx] = D[c], U[D[c]] = idx, D[c] = idx;
    ```

    强烈建议读者完全掌握这几步的顺序后再继续阅读本文。
  - 把 idx 插入到 $first(r)$ 的正右方，此时：
    - idx 右侧的结点为原来 $first(r)$ 的右结点；
    - 原来 $first(r)$ 右侧的结点的左结点为 idx；
    - idx 的左结点为 $first(r)$；
    - $first(r)$ 的右结点为 idx。

    ```
    L[idx] = first[r], R[idx] = R[first[r]];
    L[R[first[r]]] = idx, R[first[r]] = idx;
    ```

    强烈建议读者完全掌握这几步的顺序后再继续阅读本文。

`insert(r, c)` 这个操作可以通过图片来辅助理解：



留心曲线箭头的方向。

`insert(r, c)` 的代码实现如下：

```
void insert(const int &r, const int &c) {
  row[++idx] = r, col[idx] = c, ++siz[c];
  U[idx] = c, D[idx] = D[c], U[D[c]] = idx, D[c] = idx;
  if (!first[r])
    first[r] = L[idx] = R[idx] = idx;
  else {
    L[idx] = first[r], R[idx] = R[first[r]];
    L[R[first[r]]] = idx, R[first[r]] = idx;
  }
}
```

- dance 操作

`dance()` 即为递归地删除以及还原各个行列的过程。

1. 如果 号结点没有右结点，那么矩阵为空，记录答案并返回；
2. 选择列元素个数最少的一列，并删掉这一列；
3. 遍历这一列所有有 的行，枚举它是否被选择；

4. 递归调用 `dance()`，如果可行，则返回；如果不可行，则恢复被选择的行；
5. 如果无解，则返回。

`dance()` 的代码实现如下：

```
bool dance(int dep) {
  int i, j, c = R[0];
  if (!R[0]) {
    ans = dep;
    return 1;
  }
  IT(i, R, 0) if (siz[i] < siz[c]) c = i;
  remove(c);
  IT(i, D, c) {
    stk[dep] = row[i];
    IT(j, R, i) remove(col[j]);
    if (dance(dep + 1)) return 1;
    IT(j, L, i) recover(col[j]);
  }
  recover(c);
  return 0;
}
```

其中 `stk[]` 用来记录答案。

注意我们每次优先选择列元素个数最少的一列进行删除，这样能保证程序具有一定的启发性，使搜索树分支最少。

对于重复覆盖问题，在搜索时可以用估价函数（与 A* 中类似）进行剪枝：若当前最好情况下所选行数超过目前最优解，则可以直接返回。

模板

```
#include <bits/stdc++.h>
const int N = 500 + 10;
int n, m, idx, ans;
int first[N], siz[N], stk[N];

int read() {  // 快读
  int x = 0, f = 0, ch;
```

```
  while (!isdigit(ch = getchar())) f |= ch == '-';
  while (isdigit(ch)) x = (x << 1) + (x << 3) + (ch ^ 48),
ch = getchar();
  return f ? -x : x;
}

struct DLX {
  static const int MAXSIZE = 1e5 + 10;
  int n, m, tot, first[MAXSIZE + 10], siz[MAXSIZE + 10];
  int L[MAXSIZE + 10], R[MAXSIZE + 10], U[MAXSIZE + 10],
D[MAXSIZE + 10];
  int col[MAXSIZE + 10], row[MAXSIZE + 10];

  void build(const int &r, const int &c) {  // 进行build操
作
    n = r, m = c;
    for (int i = 0; i <= c; ++i) {
      L[i] = i - 1, R[i] = i + 1;
      U[i] = D[i] = i;
    }
    L[0] = c, R[c] = 0, tot = c;
    memset(first, 0, sizeof(first));
    memset(siz, 0, sizeof(siz));
  }

  void insert(const int &r, const int &c) {  // 进行insert
操作
    col[++tot] = c, row[tot] = r, ++siz[c];
    D[tot] = D[c], U[D[c]] = tot, U[tot] = c, D[c] = tot;
    if (!first[r])
      first[r] = L[tot] = R[tot] = tot;
    else {
      R[tot] = R[first[r]], L[R[first[r]]] = tot;
      L[tot] = first[r], R[first[r]] = tot;
    }
  }

  void remove(const int &c) {  // 进行remove操作
    int i, j;
    L[R[c]] = L[c], R[L[c]] = R[c];
```

```
    for (i = D[c]; i != c; i = D[i])
      for (j = R[i]; j != i; j = R[j])
        U[D[j]] = U[j], D[U[j]] = D[j], --siz[col[j]];
  }

  void recover(const int &c) {  // 进行recover操作
    int i, j;
    for (i = U[c]; i != c; i = U[i])
      for (j = L[i]; j != i; j = L[j]) U[D[j]] = D[U[j]] =
j, ++siz[col[j]];
    L[R[c]] = R[L[c]] = c;
  }

  bool dance(int dep) {  // dance
    if (!R[0]) {
      ans = dep;
      return 1;
    }
    int i, j, c = R[0];
    for (i = R[0]; i != 0; i = R[i])
      if (siz[i] < siz[c]) c = i;
    remove(c);
    for (i = D[c]; i != c; i = D[i]) {
      stk[dep] = row[i];
      for (j = R[i]; j != i; j = R[j]) remove(col[j]);
      if (dance(dep + 1)) return 1;
      for (j = L[i]; j != i; j = L[j]) recover(col[j]);
    }
    recover(c);
    return 0;
  }
} solver;

int main() {
  n = read(), m = read();
  solver.build(n, m);
  for (int i = 1; i <= n; ++i)
    for (int j = 1; j <= m; ++j) {
      int x = read();
      if (x) solver.insert(i, j);
```

```
    }
  solver.dance(1);
  if (ans)
    for (int i = 1; i < ans; ++i) printf("%d ", stk[i]);
  else
    puts("No Solution!");
  return 0;
}
```

性质

DLX 递归及回溯的次数与矩阵中 1 的个数有关，与矩阵的 r、c 等参数无关。因此，它的时间复杂度是 **指数级** 的，理论复杂度大概在 $O(c^n)$ 左右，其中 c 为某个非常接近于 1 的常数，n 为矩阵中 1 的个数。

但实际情况下 DLX 表现良好，一般能解决大部分的问题。

建模

DLX 的难点，不全在于链表的建立，而在于建模。

请确保已经完全掌握 DLX 模板后再继续阅读本文。

我们每拿到一个题，应该考虑行和列所表示的意义：

- 行表示*决策*，因为每行对应着一个集合，也就对应着选/不选；
- 列表示*状态*，因为第 列对应着某个条件 P_i。

对于某一行而言，由于不同的列的值不尽相同，我们 **由不同的状态，定义了一个决策。**

例题 1 P1784 数独

解题思路：

解题思路

先考虑决策是什么。

在这一题中，每一个决策可以用形如 (r, c, w) 的有序三元组表示。

注意到「宫」并不是决策的参数，因为它 **可以被每个确定的** (r, c) **表示。**

因此有 $9 \times 9 \times 9 = 729$ 行。

再考虑状态是什么。

我们思考一下 (r, c, w) 这个决将会造成什么影响。记 (r, c) 所在的宫为 b。

1. 第 r 行用了一个 w（用 $9 \times 9 = 81$ 列表示）；
2. 第 c 列用了一个 w（用 $9 \times 9 = 81$ 列表示）；
3. 第 b 宫用了一个 w（用 $9 \times 9 = 81$ 列表示）；
4. (r, c) 中填入了一个数（用 $9 \times 9 = 81$ 列表示）。

因此有 $81 \times 4 = 324$ 列，共 $729 \times 4 = 2916$ 个 1。

至此，我们成功地将 9×9 的数独问题转化成了一个 **有** 729 **行，** 324 **列，共** 2916 **个** 1 的精确覆盖问题。

参考代码：

```
#include <bits/stdc++.h>
const int N = 1e6 + 10;
int ans[10][10], stk[N];

int read() {
  int x = 0, f = 0, ch;
  while (!isdigit(ch = getchar())) f |= ch == '-';
  while (isdigit(ch)) x = (x << 1) + (x << 3) + (ch ^ 48),
ch = getchar();
  return f ? -x : x;
}  // 快读

struct DLX {
  static const int MAXSIZE = 1e5 + 10;
  int n, m, tot, first[MAXSIZE + 10], siz[MAXSIZE + 10];
  int L[MAXSIZE + 10], R[MAXSIZE + 10], U[MAXSIZE + 10],
D[MAXSIZE + 10];
  int col[MAXSIZE + 10], row[MAXSIZE + 10];

  void build(const int &r, const int &c) {  // 进行build操
作
    n = r, m = c;
    for (int i = 0; i <= c; ++i) {
```

```
      L[i] = i - 1, R[i] = i + 1;
      U[i] = D[i] = i;
    }
    L[0] = c, R[c] = 0, tot = c;
    memset(first, 0, sizeof(first));
    memset(siz, 0, sizeof(siz));
  }

  void insert(const int &r, const int &c) {  // 进行insert
操作
    col[++tot] = c, row[tot] = r, ++siz[c];
    D[tot] = D[c], U[D[c]] = tot, U[tot] = c, D[c] = tot;
    if (!first[r])
      first[r] = L[tot] = R[tot] = tot;
    else {
      R[tot] = R[first[r]], L[R[first[r]]] = tot;
      L[tot] = first[r], R[first[r]] = tot;
    }
  }

  void remove(const int &c) {  // 进行remove操作
    int i, j;
    L[R[c]] = L[c], R[L[c]] = R[c];
    for (i = D[c]; i != c; i = D[i])
      for (j = R[i]; j != i; j = R[j])
        U[D[j]] = U[j], D[U[j]] = D[j], --siz[col[j]];
  }

  void recover(const int &c) {  // 进行recover操作
    int i, j;
    for (i = U[c]; i != c; i = U[i])
      for (j = L[i]; j != i; j = L[j]) U[D[j]] = D[U[j]] =
j, ++siz[col[j]];
    L[R[c]] = R[L[c]] = c;
  }

  bool dance(int dep) {  // dance
    int i, j, c = R[0];
    if (!R[0]) {
      for (i = 1; i < dep; ++i) {
```

```
        int x = (stk[i] - 1) / 9 / 9 + 1;
        int y = (stk[i] - 1) / 9 % 9 + 1;
        int v = (stk[i] - 1) % 9 + 1;
        ans[x][y] = v;
      }
      return 1;
    }
    for (i = R[0]; i != 0; i = R[i])
      if (siz[i] < siz[c]) c = i;
    remove(c);
    for (i = D[c]; i != c; i = D[i]) {
      stk[dep] = row[i];
      for (j = R[i]; j != i; j = R[j]) remove(col[j]);
      if (dance(dep + 1)) return 1;
      for (j = L[i]; j != i; j = L[j]) recover(col[j]);
    }
    recover(c);
    return 0;
  }
} solver;

int GetId(int row, int col, int num) {
  return (row - 1) * 9 * 9 + (col - 1) * 9 + num;
}

void Insert(int row, int col, int num) {
  int dx = (row - 1) / 3 + 1;
  int dy = (col - 1) / 3 + 1;
  int room = (dx - 1) * 3 + dy;
  int id = GetId(row, col, num);
  int f1 = (row - 1) * 9 + num;             // task 1
  int f2 = 81 + (col - 1) * 9 + num;        // task 2
  int f3 = 81 * 2 + (room - 1) * 9 + num;   // task 3
  int f4 = 81 * 3 + (row - 1) * 9 + col;    // task 4
  solver.insert(id, f1);
  solver.insert(id, f2);
  solver.insert(id, f3);
  solver.insert(id, f4);
}
```

```
int main() {
  solver.build(729, 324);
  for (int i = 1; i <= 9; ++i)
    for (int j = 1; j <= 9; ++j) {
      ans[i][j] = read();
      for (int v = 1; v <= 9; ++v) {
        if (ans[i][j] && ans[i][j] != v) continue;
        Insert(i, j, v);
      }
    }
  solver.dance(1);
  for (int i = 1; i <= 9; ++i, putchar('\n'))
    for (int j = 1; j <= 9; ++j, putchar(' '))
printf("%d", ans[i][j]);
  return 0;
}
```

例题 2 靶形数独

解题思路：

这一题与 数独 的模型构建 **一模一样**，主要区别在于答案的更新。

这一题可以开一个权值数组，每次找到一组数独的解时，

每个位置上的数乘上对应的权值计入答案即可。

参考代码：

```
#include <bits/stdc++.h>
const int oo = 0x3f3f3f3f;
const int N = 1e5 + 10;
const int e[] = {6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 7, 7, 7, 7,
7, 7, 7, 6, 6, 7,  8,
                 8, 8, 8, 8, 7, 6, 6, 7, 8, 9, 9, 9, 8, 7,
6, 6, 7, 8, 9, 10, 9,
                 8, 7, 6, 6, 7, 8, 9, 9, 9, 8, 7, 6, 6, 7,
8, 8, 8, 8, 8, 7,  6,
                 6, 7, 7, 7, 7, 7, 7, 7, 6, 6, 6, 6, 6, 6,
6, 6, 6, 6};
int ans = -oo, a[10][10], stk[N];
```

```
int read() {
  int x = 0, f = 0, ch;
  while (!isdigit(ch = getchar())) f |= ch == '-';
  while (isdigit(ch)) x = (x << 1) + (x << 3) + (ch ^ 48),
ch = getchar();
  return f ? -x : x;
}

int GetWeight(int row, int col, int num) {  // 求数乘上对应的
权值
  return num * e[(row - 1) * 9 + (col - 1)];
}

struct DLX {
  static const int MAXSIZE = 1e5 + 10;
  int n, m, tot, first[MAXSIZE + 10], siz[MAXSIZE + 10];
  int L[MAXSIZE + 10], R[MAXSIZE + 10], U[MAXSIZE + 10],
D[MAXSIZE + 10];
  int col[MAXSIZE + 10], row[MAXSIZE + 10];

  void build(const int &r, const int &c) {  // 进行build操
作
    n = r, m = c;
    for (int i = 0; i <= c; ++i) {
      L[i] = i - 1, R[i] = i + 1;
      U[i] = D[i] = i;
    }
    L[0] = c, R[c] = 0, tot = c;
    memset(first, 0, sizeof(first));
    memset(siz, 0, sizeof(siz));
  }

  void insert(const int &r, const int &c) {  // 进行insert
操作
    col[++tot] = c, row[tot] = r, ++siz[c];
    D[tot] = D[c], U[D[c]] = tot, U[tot] = c, D[c] = tot;
    if (!first[r])
      first[r] = L[tot] = R[tot] = tot;
    else {
```

```
      R[tot] = R[first[r]], L[R[first[r]]] = tot;
      L[tot] = first[r], R[first[r]] = tot;
    }
  }

  void remove(const int &c) {  // 进行remove操作
    int i, j;
    L[R[c]] = L[c], R[L[c]] = R[c];
    for (i = D[c]; i != c; i = D[i])
      for (j = R[i]; j != i; j = R[j])
        U[D[j]] = U[j], D[U[j]] = D[j], --siz[col[j]];
  }

  void recover(const int &c) {  // 进行recover操作
    int i, j;
    for (i = U[c]; i != c; i = U[i])
      for (j = L[i]; j != i; j = L[j]) U[D[j]] = D[U[j]] =
j, ++siz[col[j]];
    L[R[c]] = R[L[c]] = c;
  }

  void dance(int dep) {  // dance
    int i, j, c = R[0];
    if (!R[0]) {
      int cur_ans = 0;
      for (i = 1; i < dep; ++i) {
        int cur_row = (stk[i] - 1) / 9 / 9 + 1;
        int cur_col = (stk[i] - 1) / 9 % 9 + 1;
        int cur_num = (stk[i] - 1) % 9 + 1;
        cur_ans += GetWeight(cur_row, cur_col, cur_num);
      }
      ans = std::max(ans, cur_ans);
      return;
    }
    for (i = R[0]; i != 0; i = R[i])
      if (siz[i] < siz[c]) c = i;
    remove(c);
    for (i = D[c]; i != c; i = D[i]) {
      stk[dep] = row[i];
      for (j = R[i]; j != i; j = R[j]) remove(col[j]);
```

```
      dance(dep + 1);
      for (j = L[i]; j != i; j = L[j]) recover(col[j]);
    }
    recover(c);
  }
} solver;

int GetId(int row, int col, int num) {
  return (row - 1) * 9 * 9 + (col - 1) * 9 + num;
}

void Insert(int row, int col, int num) {
  int dx = (row - 1) / 3 + 1;     // r
  int dy = (col - 1) / 3 + 1;     // c
  int room = (dx - 1) * 3 + dy;  // room
  int id = GetId(row, col, num);
  int f1 = (row - 1) * 9 + num;             // task 1
  int f2 = 81 + (col - 1) * 9 + num;        // task 2
  int f3 = 81 * 2 + (room - 1) * 9 + num;  // task 3
  int f4 = 81 * 3 + (row - 1) * 9 + col;   // task 4
  solver.insert(id, f1);
  solver.insert(id, f2);
  solver.insert(id, f3);
  solver.insert(id, f4);
}

int main() {
  solver.build(729, 324);
  for (int i = 1; i <= 9; ++i)
    for (int j = 1; j <= 9; ++j) {
      a[i][j] = read();
      for (int v = 1; v <= 9; ++v) {
        if (a[i][j] && v != a[i][j]) continue;
        Insert(i, j, v);
      }
    }
  solver.dance(1);
  printf("%d", ans == -oo ? -1 : ans);
  return 0;
}
```

例题 3 「NOI2005」智慧珠游戏

解题思路

定义：题中给我们的智慧珠的形态，称为这个智慧珠的*标准形态*。

显然，我们可以通过改变两个参数 d（表示顺时针旋转 90° 的次数）和 f（是否水平翻转）来改变这个智慧珠的形态。

仍然，我们先考虑决策是什么。

在这一题中，每一个决策可以用形如 (v,d,f,i) 的有序五元组表示。

表示第 i 个智慧珠的*标准形态*的左上角的位置，序号为 v，经过了 d 次顺时针转 90°。

巧合的是，我们可以令 $f=1$ 时不水平翻转，$f=-1$ 时水平翻转，从而达到简化代码的目的。

因此有 $55\times 4\times 2\times 12=5280$ 行。

需要注意的是，因为一些不合法的填充，如 $(1,0,1,4)$，

所以 **在实际操作中，空的智慧珠棋盘也只需要建出 2730 行。**

再考虑状态是什么。

这一题的状态比较简单。

我们思考一下，(v,d,f,i) 这个决策会造成什么影响。

1. 某些格子被占了（用 55 列表示）；
2. 第 i 个智慧珠被用了（用 12 列表示）。

因此有 $55+12=67$ 列，共 $5280\times(5+1)=31680$ 个 1。

至此，我们成功地将智慧珠游戏转化成了一个 **有 5280 行，67 列，共 31680 个 1** 的精确覆盖问题。

参考代码：

```
#include <bits/stdc++.h>
int numcol, numrow;
int dfn[3000], tx[2], nxt[2], num[50][50], vis[50];
char ans[50][50];
const int f[2] = {-1, 1};
const int table[12][5][2] = {
    // directions of shapes
    {{0, 0}, {1, 0}, {0, 1}},                   // A
    {{0, 0}, {0, 1}, {0, 2}, {0, 3}},           // B
    {{0, 0}, {1, 0}, {0, 1}, {0, 2}},           // C
    {{0, 0}, {1, 0}, {0, 1}, {1, 1}},           // D
    {{0, 0}, {1, 0}, {2, 0}, {2, 1}, {2, 2}},   // E
    {{0, 0}, {0, 1}, {1, 1}, {0, 2}, {0, 3}},   // F
    {{0, 0}, {1, 0}, {0, 1}, {0, 2}, {1, 2}},   // G
    {{0, 0}, {1, 0}, {0, 1}, {1, 1}, {0, 2}},   // H
    {{0, 0}, {0, 1}, {0, 2}, {1, 2}, {1, 3}},   // I
```

```
    {{0, 0}, {-1, 1}, {0, 1}, {1, 1}, {0, 2}},  // J
    {{0, 0}, {1, 0}, {1, 1}, {2, 1}, {2, 2}},   // K
    {{0, 0}, {1, 0}, {0, 1}, {0, 2}, {0, 3}},   // L
};
const int len[12] = {3, 4, 4, 4, 5, 5, 5, 5, 5, 5, 5, 5};
const int getx[] = {0,  1,  2,  2,  3,  3,  3,  4,  4,  4,
 4,  5,  5,  5,  5,
                    5,  6,  6,  6,  6,  6,  6,  7,  7,  7,
 7,  7,  7,  7,  8,
                    8,  8,  8,  8,  8,  8,  8,  9,  9,  9,
 9,  9,  9,  9,  9,
                    9,  10, 10, 10, 10, 10, 10, 10, 10,
10, 10, 11, 11, 11, 11,
                    11, 11, 11, 11, 11, 11, 11, 12, 12,
12, 12, 12, 12, 12, 12,
                    12, 12, 12, 12, 13, 13, 13, 13, 13,
13, 13, 13, 13, 13, 13,
                    13, 13, 14, 14, 14, 14, 14, 14, 14,
14, 14};
const int gety[] = {0, 1, 1, 2,  1,  2,  3,  1, 2,  3,  4,
 1, 2, 3, 4,  5,  1,
                    2, 3, 4, 5,  6,  1,  2,  3, 4,  5,  6,
 7, 1, 2, 3,  4,  5,
                    6, 7, 8, 1,  2,  3,  4,  5, 6,  7,  8,
 9, 1, 2, 3,  4,  5,
                    6, 7, 8, 9,  10, 1,  2,  3, 4,  5,  6,
 7, 8, 9, 10, 11, 1,
                    2, 3, 4, 5,  6,  7,  8,  9, 10, 11,
12, 1, 2, 3, 4,  5,  6,
                    7, 8, 9, 10, 11, 12, 13, 1, 2,  3,  4,
 5, 6, 7, 8,  9};

struct DLX {
  static const int MS = 1e5 + 10;
  int n, m, tot, first[MS], siz[MS];
  int L[MS], R[MS], U[MS], D[MS];
  int col[MS], row[MS];

  void build(const int &r, const int &c) {
    n = r, m = c;
```

```
    for (int i = 0; i <= c; ++i) {
      L[i] = i - 1, R[i] = i + 1;
      U[i] = D[i] = i;
    }
    L[0] = c, R[c] = 0, tot = c;
    memset(first, 0, sizeof(first));
    memset(siz, 0, sizeof(siz));
  }

  void insert(const int &r, const int &c) {  // insert
    col[++tot] = c, row[tot] = r, ++siz[c];
    D[tot] = D[c], U[D[c]] = tot, U[tot] = c, D[c] = tot;
    if (!first[r])
      first[r] = L[tot] = R[tot] = tot;
    else
      R[tot] = R[first[r]], L[R[first[r]]] = tot, L[tot] =
first[r],
      R[first[r]] = tot;  // !
  }

  void remove(const int &c) {  // remove
    int i, j;
    L[R[c]] = L[c], R[L[c]] = R[c];
    for (i = D[c]; i != c; i = D[i])
      for (j = R[i]; j != i; j = R[j])
        U[D[j]] = U[j], D[U[j]] = D[j], --siz[col[j]];
  }

  void recover(const int &c) {  // recover
    int i, j;
    for (i = U[c]; i != c; i = U[i])
      for (j = L[i]; j != i; j = L[j]) U[D[j]] = D[U[j]] =
j, ++siz[col[j]];
    L[R[c]] = R[L[c]] = c;
  }

  bool dance() {  // dance
    if (!R[0]) return 1;
    int i, j, c = R[0];
    for (i = R[0]; i != 0; i = R[i])
```

```
      if (siz[i] < siz[c]) c = i;
    remove(c);
    for (i = D[c]; i != c; i = D[i]) {
      if (col[i] <= 55) ans[getx[col[i]]][gety[col[i]]] =
dfn[row[i]] + 'A';
      for (j = R[i]; j != i; j = R[j]) {
        remove(col[j]);
        if (col[j] <= 55) ans[getx[col[j]]][gety[col[j]]]
= dfn[row[j]] + 'A';
      }
      if (dance()) return 1;
      for (j = L[i]; j != i; j = L[j]) recover(col[j]);
    }
    recover(c);
    return 0;
  }
} solver;

int main() {
  for (int i = 1; i <= 10; ++i) scanf("%s", ans[i] + 1);
  for (int i = 1; i <= 10; ++i)
    for (int j = 1; j <= i; ++j) {
      if (ans[i][j] != '.') vis[ans[i][j] - 'A'] = 1;
      num[i][j] = ++numcol;
    }
  solver.build(2730, numcol + 12);
  /*******build*******/
  for (int id = 0, op; id < 12; ++id) {  // every block
    for (++numcol, op = 0; op <= 1; ++op) {
      for (int dx = 0; dx <= 1; ++dx) {
        for (int dy = 0; dy <= 1; ++dy) {
          for (tx[0] = 1; tx[0] <= 10; ++tx[0]) {
            for (tx[1] = 1; tx[1] <= tx[0]; ++tx[1]) {
              bool flag = 1;
              for (int k = 0; k < len[id]; ++k) {
                nxt[op] = tx[op] + f[dx] * table[id][k]
[0];
                nxt[op ^ 1] = tx[op ^ 1] + f[dy] *
table[id][k][1];
                if (vis[id]) {
```

```
                    if (ans[nxt[0]][nxt[1]] != id + 'A') {
                      flag = 0;
                      break;
                    }
                  } else if (ans[nxt[0]][nxt[1]] != '.') {
                    flag = 0;
                    break;
                  }
                }
                if (!flag) continue;
                dfn[++numrow] = id;
                solver.insert(numrow, numcol);
                for (int k = 0; k < len[id]; ++k) {
                  nxt[op] = tx[op] + f[dx] * table[id][k]
[0];
                  nxt[op ^ 1] = tx[op ^ 1] + f[dy] *
table[id][k][1];
                  solver.insert(numrow, num[nxt[0]]
[nxt[1]]);
                }
              }
            }
          }
        }
      }
    }
  }
  /********end********/
  if (!solver.dance())
    puts("No solution");
  else
    for (int i = 1; i <= 10; ++i, puts(""))
      for (int j = 1; j <= i; ++j) putchar(ans[i][j]);
  return 0;
}
```

习题

- SUDOKU - Sudoku
- 「kuangbin 带你飞」专题三 Dancing Links

外部链接

- 夜深人静写算法（九）- Dancing Links X（跳舞链）_WhereIsHeroFrom 的博客》
- 跳跃的舞者，舞蹈链（Dancing Links）算法——求解精确覆盖问题 - 万仓一黍
- DLX 算法一览 - zhangjianjunab
- 搜索：DLX 算法 - 静听风吟。
- 《算法竞赛入门经典 - 训练指南》

Alpha-Beta 剪枝

此页面将简要介绍 minimax 算法和 $\alpha - \beta$ 剪枝。

Minimax 算法

定义

Minimax 算法又叫极小化极大算法，是一种找出失败的最大可能性中的最小值的算法。1

在局面确定的双人对弈里，常进行对抗搜索，构建一棵每个节点都为一个确定状态的搜索树。奇数层为己方先手，偶数层为对方先手。搜索树上每个叶子节点都会被赋予一个估值，估值越大代表我方赢面越大。我方追求更大的赢面，而对方会设法降低我方的赢面，体现在搜索树上就是，奇数层节点（我方节点）总是会选择赢面最大的子节点状态，而偶数层（对方节点）总是会选择我方赢面最小的的子节点状态。

过程

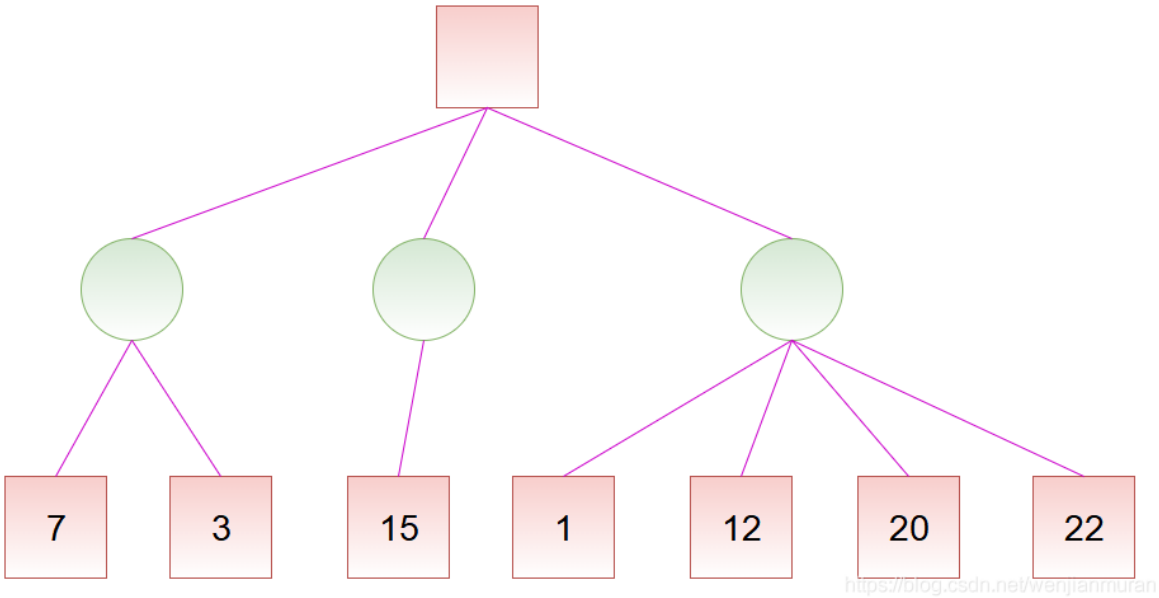
Minimax 算法的整个过程，会从上到下遍历搜索树，回溯时利用子树信息更新答案，最后得到根节点的值，意义就是我方在双方都采取最优策略下能获得的最大分数。

解释

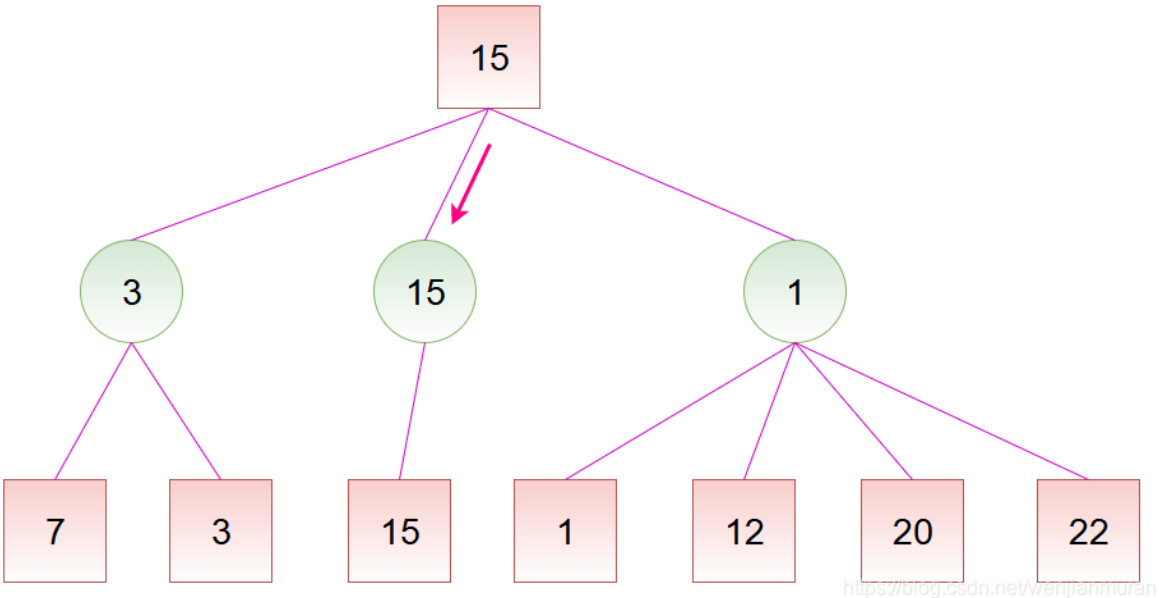
来看一个简单的例子。

称我方为 MAX，对方为 MIN，图示如下：

例如，对于如下的局势，假设从左往右搜索，根节点的数值为我方赢面：



我方应选择中间的路线。因为，如果选择左边的路线，最差的赢面是 3；如果选择中间的路线，最差的赢面是 15；如果选择右边的路线，最差的赢面是 1。虽然选择右边的路线可能有 22 的赢面，但对方也可能使我方只有 1 的赢面，假设对方会选择使得我方赢面最小的方向走，那么经过权衡，显然选择中间的路线更为稳妥。



实际上，在看右边的路线时，当发现赢面可能为 1 就不必再去看赢面为 12、20、22 的分支了，因为已经可以确定右边的路线不是最好的。

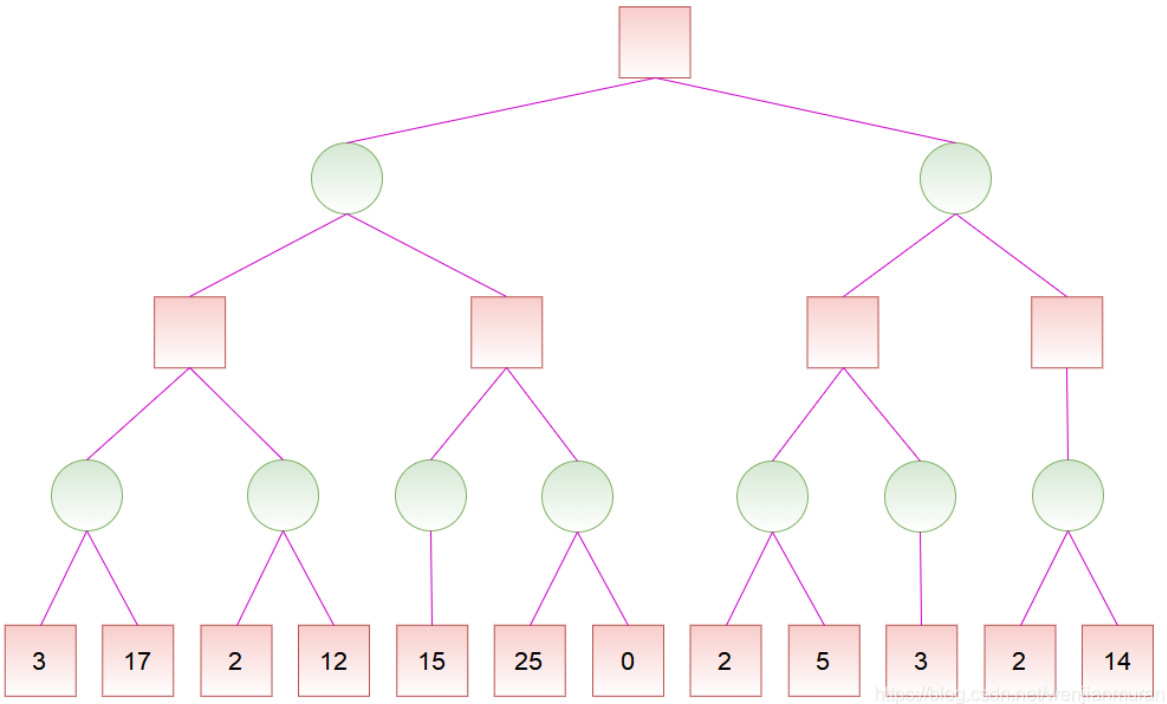
朴素的 Minimax 算法常常需要构建一棵庞大的搜索树，时间和空间复杂度都将不能承受。而 $\alpha - \beta$ 剪枝就是利用搜索树每个节点取值的上下界来对 Minimax 进行剪枝优化的一种方法。

需要注意的是，对于不同的问题，搜索树每个节点上的值有着不同的含义，它可以是估值、分数、赢的概率等等，为方便起见，我们下面统一用分数来称呼。

alpha-beta 剪枝

过程

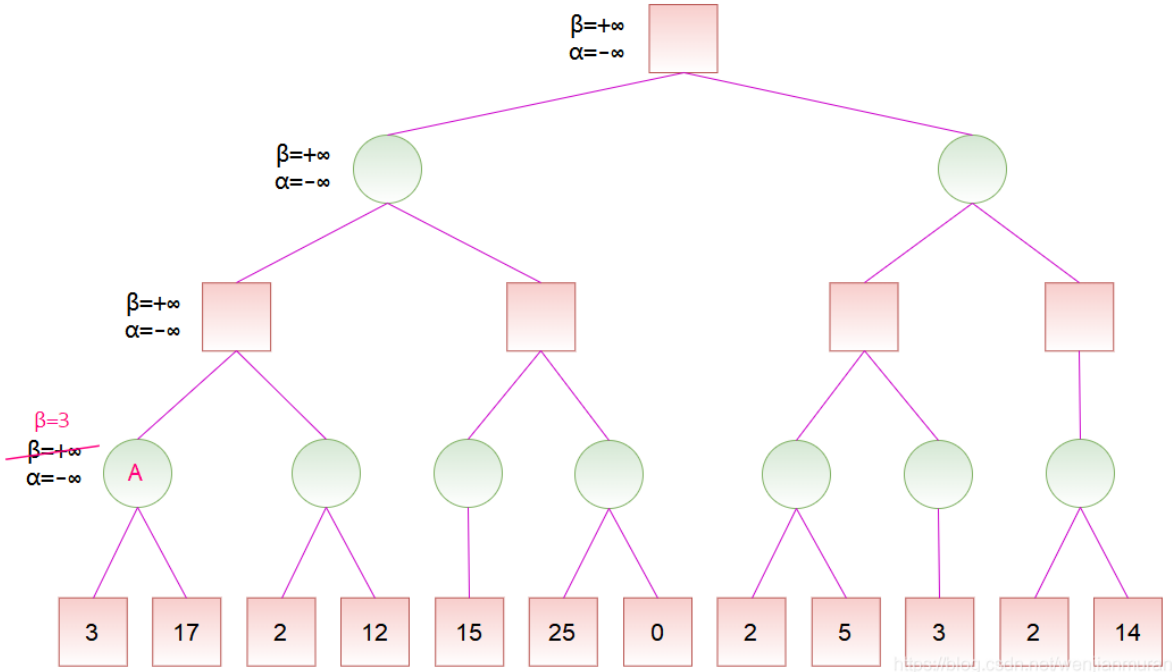
对于如下的局势，假设从左往右搜索：



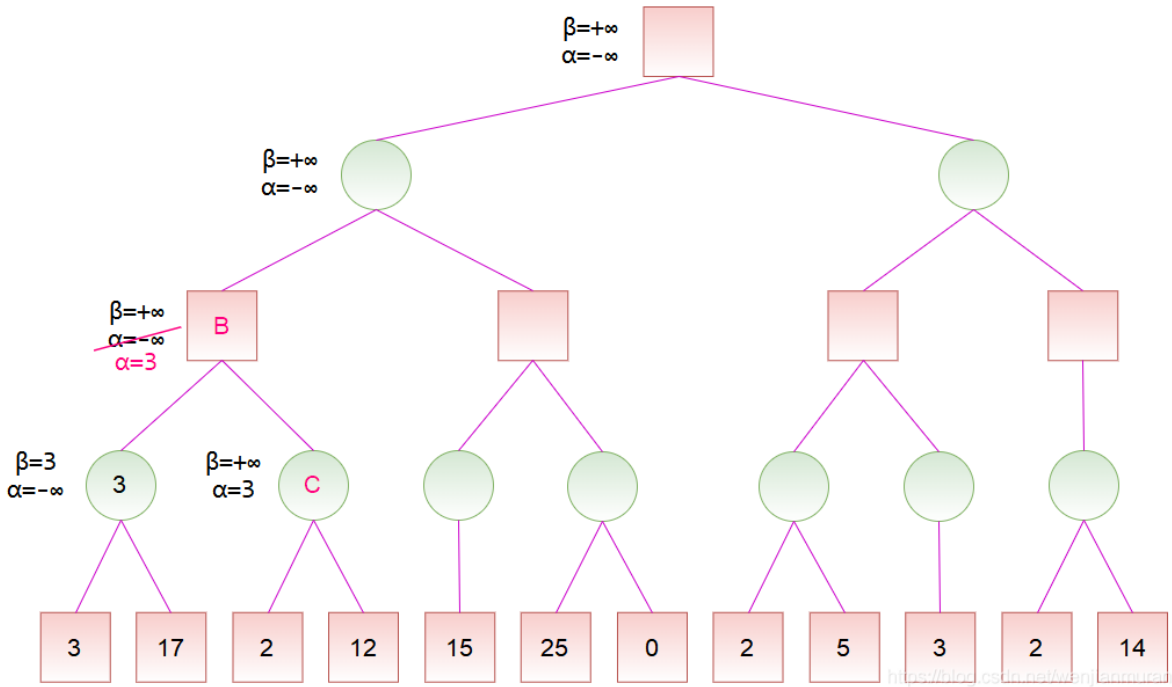
若已知某节点的所有子节点的分数，则可以算出该节点的分数：对于 MAX 节点，取最大分数；对于 MIN 节点，取最小分数。

若已知某节点的部分子节点的分数，虽然不能算出该节点的分数，但可以算出该节点的分数的取值范围。同时，利用该节点的分数的取值范围，在搜素其子节点时，如果已经确定没有更好的走法，就不必再搜索剩余的子节点了。

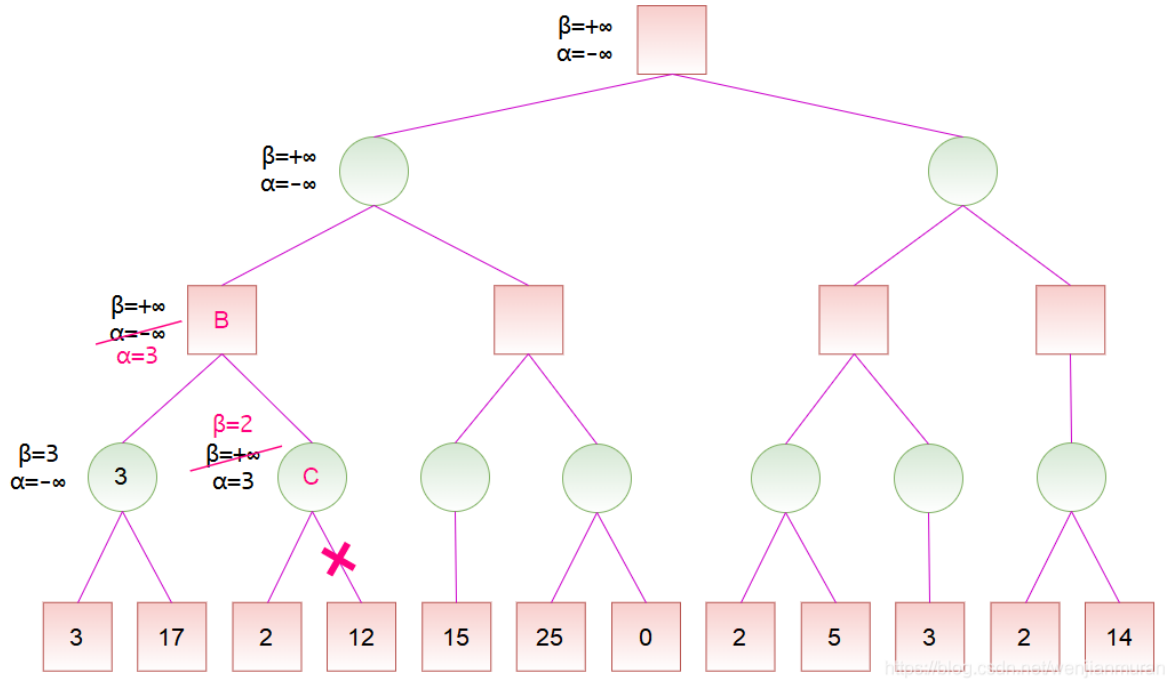
记 v 为节点的分数，且 $\alpha \leq v \leq \beta$，即 为最大下界，β 为最小上界。当 $\alpha \geq \beta$ 时，该节点剩余的分支就不必继续搜索了（也就是可以进行剪枝了）。注意，当 $\alpha = \beta$ 时，也可以剪枝，这是因为不会有更好的结果了，但可能有更差的结果。



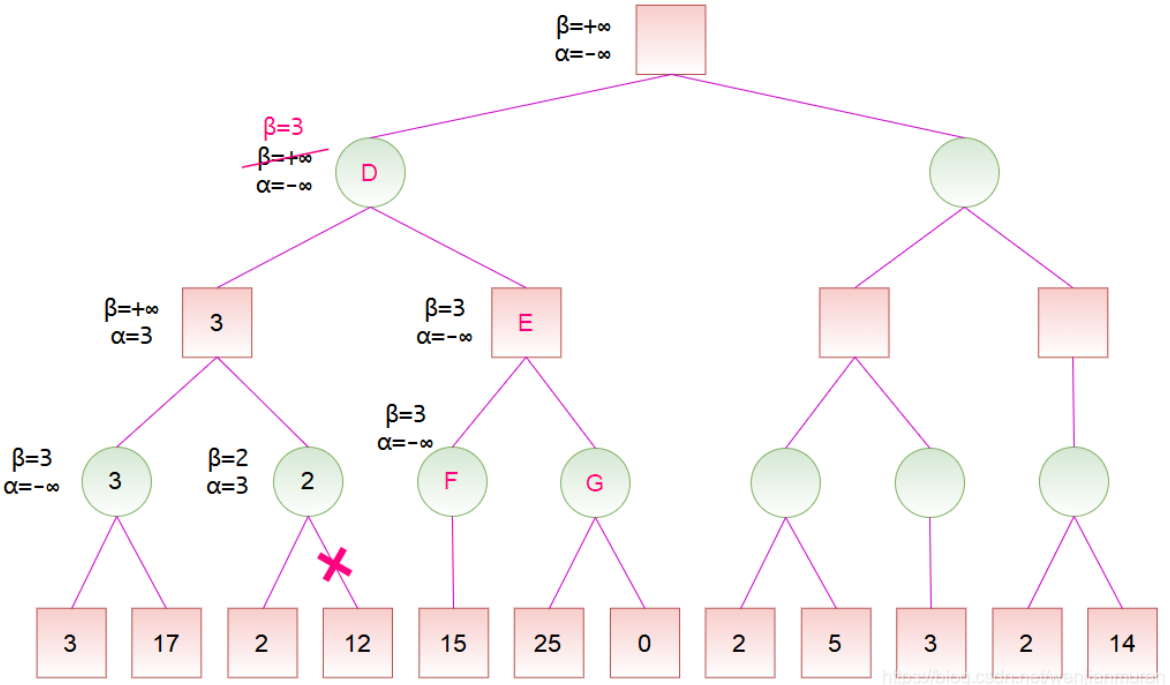
初始化时，令 $\alpha = -\infty, \beta = +\infty$，也就是 $-\infty \leq v \leq +\infty$。到节点 A 时，由于左子节点的分数为 3，而节点 A 是 MIN 节点，试图找分数小的走法，于是将 β 值修改为 3，这是因为 3 小于当前的 β 值（$\beta = +\infty$）。然后节点 A 的右子节点的分数为 17，此时不修改节点 A 的 β 值，这是因为 17 大于当前的 β 值（$\beta = 3$）。之后，节点 A 的所有子节点搜索完毕，即可计算出节点 A 的分数为 3。



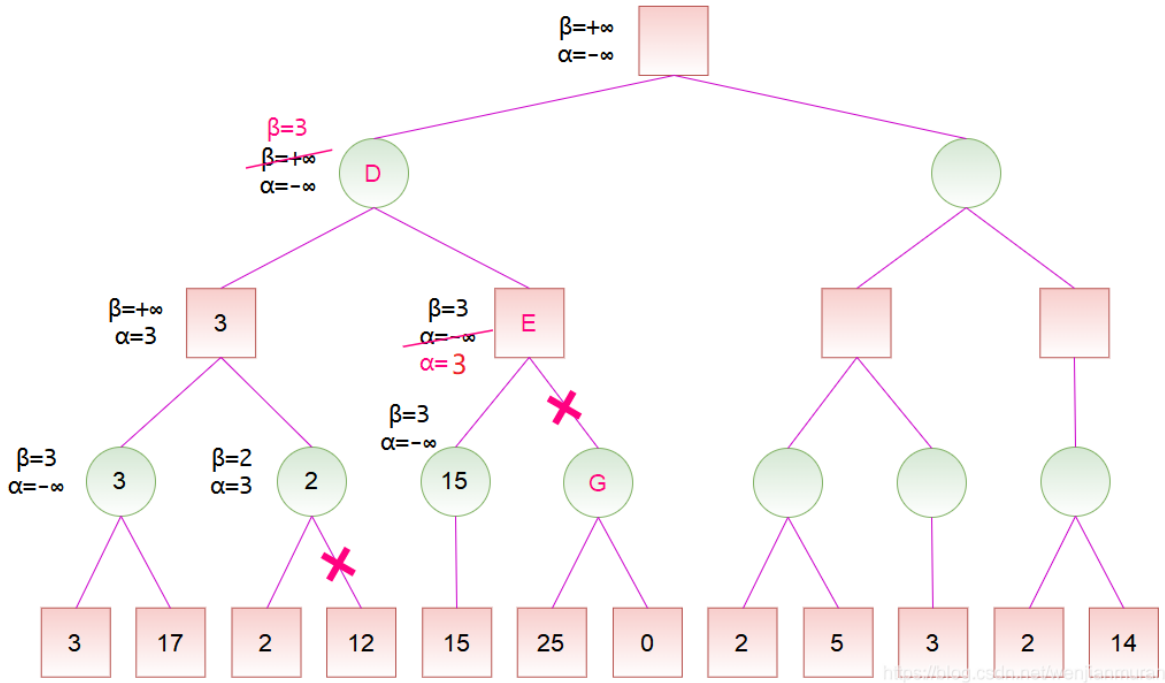
节点 A 是节点 B 的子节点，计算出节点 A 的分数后，可以更新节点 B 的分数范围。由于节点 B 是 MAX 节点，试图找分数大的走法，于是将 α 值修改为 3，这是因为 3 大于当前的 α 值（$\alpha = -\infty$）。之后搜索节点 B 的右子节点 C，并将节点 B 的 α 和 β 值传递给节点 C。



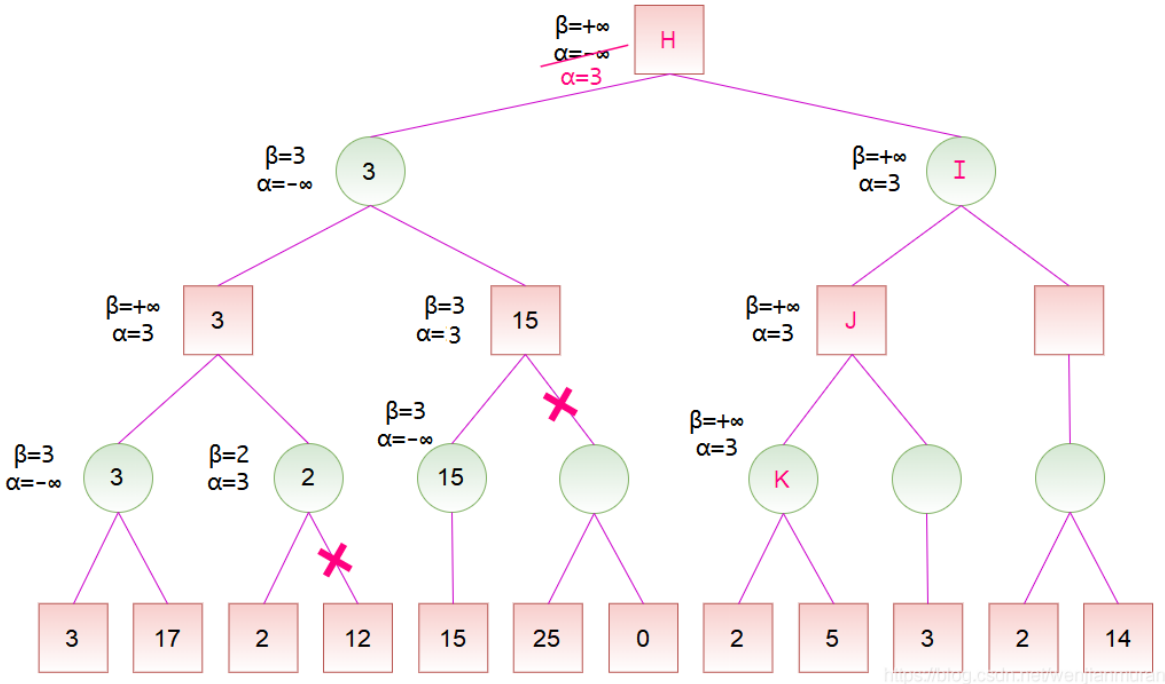
对于节点 C，由于左子节点的分数为 2，而节点 C 是 MIN 节点，于是将 β 值修改为 2。此时 $\alpha \geq \beta$，故节点 C 的剩余子节点就不必搜索了，因为可以确定，通过节点 C 并没有更好的走法。然后，节点 C 是 MIN 节点，将节点 C 的分数设为 β，也就是 2。由于节点 B 的所有子节点搜索完毕，即可计算出节点 B 的分数为 3。



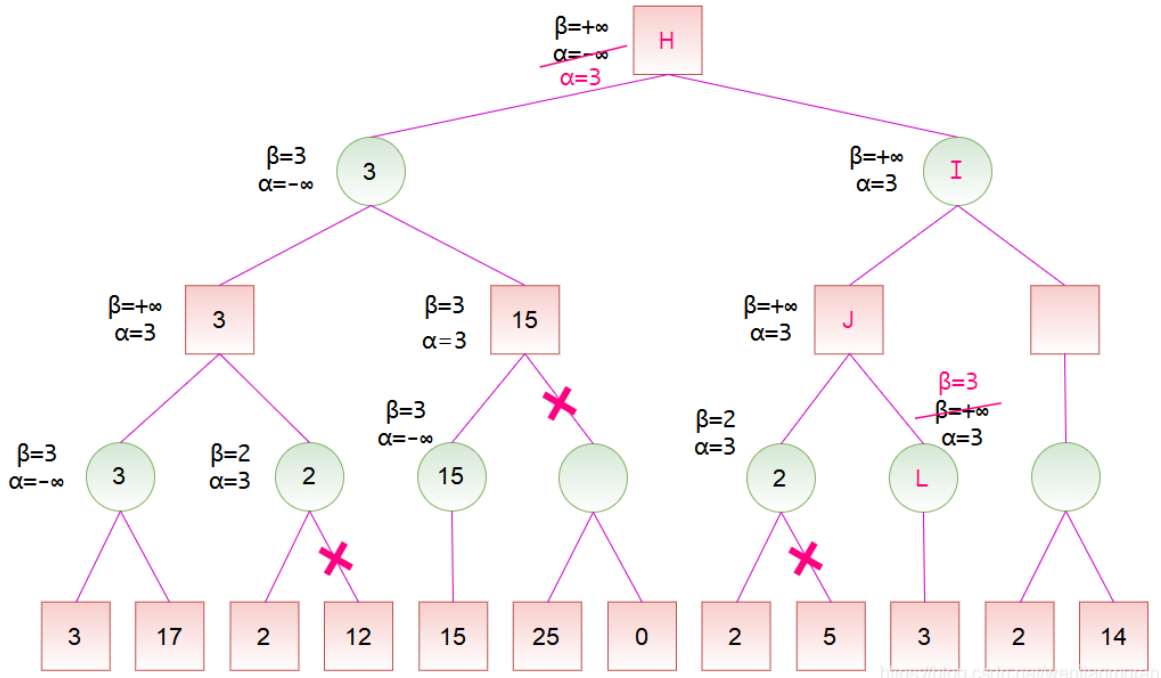
计算出节点 B 的分数后，节点 B 是节点 D 的一个子节点，故可以更新节点 D 的分数范围。由于节点 D 是 MIN 节点，于是将 β 值修改为 3。然后节点 D 将 α 和 β 值传递给节点 E，节点 E 又传递给节点 F。对于节点 F，它只有一个分数为 15 的子节点，由于 15 大于当前的 β 值，而节点 F 为 MIN 节点，所以不更新其 β 值，然后可以计算出节点 F 的分数为 15。



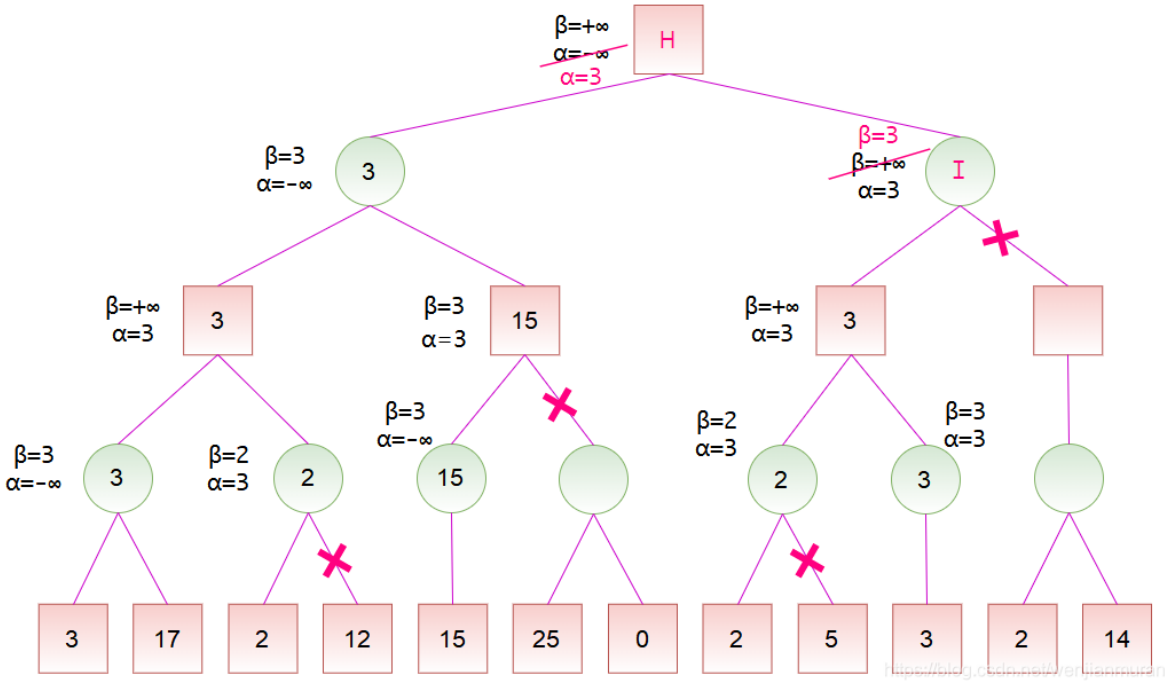
计算出节点 F 的分数后，节点 F 是节点 E 的一个子节点，故可以更新节点 E 的分数范围。节点 E 是 MAX 节点，更新 α，此时 $\alpha \geq \beta$，故可以剪去节点 E 的余下分支。然后，节点 E 是 MAX 节点，将节点 E 的分数设为 α，也就是 15。此时，节点 D 的所有子节点搜索完毕，即可计算出节点 D 的分数为 3。



计算出节点 D 的分数后，节点 D 是节点 H 的一个子节点，故可以更新节点 H 的分数范围。节点 H 是 MAX 节点，更新 α。然后，按搜索顺序，将节点 H 的 α 和 β 值依次传递给节点 I、J、K。对于节点 K，其左子节点的分数为 2，而节点 K 是 MIN 节点，更新 β，此时 $\alpha \geq \beta$，故可以剪去节点 K 的余下分支。然后，将节点 K 的分数设为 2。



计算出节点 K 的分数后，节点 K 是节点 J 的一个子节点，故可以更新节点 J 的分数范围。节点 J 是 MAX 节点，更新 α，但是，由于节点 K 的分数小于 α，所以节点 J 的 α 值维持 3 保持不变。然后，将节点 J 的 α 和 β 值传递给节点 L。由于节点 L 是 MIN 节点，更新 $\beta = 3$，此时 $\alpha \geq \beta$，故可以剪去节点 L 的余下分支，由于节点 L 没有余下分支，所以此处并没有实际剪枝。然后，将节点 L 的分数设为 3。



实现

```
int alpha_beta(int u, int alph, int beta, bool is_max) {
  if (!son_num[u]) return val[u];
  if (is_max) {
    for (int i = 0; i < son_num[u]; ++i) {
```

```
      int d = son[u][i];
      alph = max(alph, alpha_beta(d, alph, beta, is_max ^
1));
      if (alph >= beta) break;
    }
    return alph;
  } else {
    for (int i = 0; i < son_num[u]; ++i) {
      int d = son[u][i];
      beta = min(beta, alpha_beta(d, alph, beta, is_max ^
1));
      if (alph >= beta) break;
    }
    return beta;
  }
}
```

参考资料与注释

本文部分引用自博文 详解 Minimax 算法与α-β剪枝_文剑木然，遵循 CC 4.0 BY-SA 版权协议。

优化

前言

DFS（深度优先搜索）是一种常见的算法，大部分的题目都可以用 DFS 解决，但是大部分情况下，这都是骗分算法，很少会有爆搜为正解的题目。因为 DFS 的时间复杂度特别高。（没学过 DFS 的请自行补上这一课）

既然不能成为正解，那就多骗一点分吧。那么这一篇文章将介绍一些实用的优化算法（俗称「剪枝」）。

先来一段深搜模板，之后的模板将在此基础上进行修改。

```
int ans = 最坏情况, now;  // now 为当前答案

void dfs(传入数值) {
  if (到达目的地) ans = 从当前解与已有解中选最优;
  for (遍历所有可能性)
    if (可行) {
      进行操作;
      dfs(缩小规模);
      撤回操作;
    }
}
```

其中的 ans 可以是解的记录，那么从当前解与已有解中选最优就变成了输出解。

剪枝方法

最常用的剪枝有三种，记忆化搜索、最优性剪枝、可行性剪枝。

记忆化搜索

因为在搜索中，相同的传入值往往会带来相同的解，那我们就可以用数组来记忆，详见 记忆化搜索。

模板：

```
int g[MAXN];  // 定义记忆化数组
int ans = 最坏情况, now;

void dfs f(传入数值) {
  if (g[规模] != 无效数值) return;  // 或记录解，视情况而定
  if (到达目的地) ans = 从当前解与已有解中选最优;  // 输出解，视情况而定
  for (遍历所有可能性)
    if (可行) {
      进行操作;
      dfs(缩小规模);
      撤回操作;
    }
}
```

```
int main() {
  // ...
  memset(g, 无效数值, sizeof(g));  // 初始化记忆化数组
  // ...
}
```

最优性剪枝

在搜索中导致运行慢的原因还有一种，就是在当前解已经比已有解差时仍然在搜索，那么我们只需要判断一下当前解是否已经差于已有解。

模板：

```
int ans = 最坏情况, now;

void dfs(传入数值) {
  if (now比ans的答案还要差) return;
  if (到达目的地) ans = 从当前解与已有解中选最优;
  for (遍历所有可能性)
    if (可行) {
      进行操作;
      dfs(缩小规模);
      撤回操作;
    }
}
```

可行性剪枝

在搜索过程中当前解已经不可用了还继续搜索下去也是运行慢的原因。

模板：

```
int ans = 最坏情况, now;

void dfs(传入数值) {
  if (当前解已不可用) return;
  if (到达目的地) ans = 从当前解与已有解中选最优;
  for (遍历所有可能性)
    if (可行) {
      进行操作;
      dfs(缩小规模);
      撤回操作;
    }
}
```

剪枝思路

剪枝思路有很多种，大多需要对于具体问题来分析，在此简要介绍几种常见的剪枝思路。

- 极端法：考虑极端情况，如果最极端（最理想）的情况都无法满足，那么肯定实际情况搜出来的结果不会更优了。
- 调整法：通过对子树的比较剪掉重复子树和明显不是最有「前途」的子树。
- 数学方法：比如在图论中借助连通分量，数论中借助模方程的分析，借助不等式的放缩来估计下界等等。

例题

工作分配问题

题目描述

有 n 份工作要分配给 n 个人来完成，每个人完成一份。第 i 个人完成第 k 份工作所用的时间为一个正整数 $t_{i,k}$，其中 $1 \leq i,k \leq n$。试确定一个分配方案，使得完成这 n 份工作的时间总和最小。

输入包含 $n+1$ 行。

第 1 行为一个正整数 n。

第 2 行到第 $n+1$ 行中每行都包含 n 个正整数，形成了一个 $n \times n$ 的矩阵。在该矩阵中，第 i 行第 k 列元素 $t_{i,k}$ 表示第 i 个人完成第 k 件工作所要用的时间。

输出包含一个正整数，表示所有分配方案中最小的时间总和。

数据范围

$1 \leq n \leq 15$

$1 \leq t_{i,k} \leq 10^4$

输入样例

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

输出样例

```
5
```

由于每个人都必须分配到工作，在这里可以建一个二维数组 `time[i][j]`，用以表示 i 个人完成 j 号工作所花费的时间。给定一个循环，从第 1 个人开始循环分配工作，直到所有人都分配到。为第 i 个人分配工作时，再循环检查每个工作是否已被分配，没有则分配给 i 个人，否则检查下一个工作。可以用一个一维数组 `is_working[j]` 来表示第 j 号工作是否已被分配，未分配则 `is_working[j]=0`，否则 `is_working[j]=1`。利用回溯思想，在工人循环结束后回到上一工人，取消此次分配的工作，而去分配下一工作直到可以分配为止。这样，一直回溯到第 1 个工人后，就能得到所有的可行解。

检查工作分配，其实就是判断取得可行解时的二维数组的第一维下标各不相同并且第二维下标各不相同。而我们是要得到完成这 n 份工作的最小时间总和，即可行解中时间总和最小的一个，故需要再定义一个全局变量 `cost_time_total_min` 表示目前找到的解中最小的时间总和，初始 `cost_time_total_min` 为 `time[i][i]` 之和，即对角线工作时间相加之和。在所有人分配完工作时，比较 `count` 与 `cost_time_total_min` 的大小，如果 `count` 小于 `cost_time_total_min`，说明找到了一个最优解，此时就把 `count` 赋给 `cost_time_total_min`。

但考虑到算法的效率，这里还有一个剪枝优化的工作可以做。就是在每次计算局部费用变量 `count` 的值时，如果判断 `count` 已经大于 `cost_time_total_min`，就没必要再往下分配了，因为这时得到的解必然不是最优解。