搜索算法

位运算优化

河南省实验中学信息技术组

2025 年 03 月 26 日

知识回顾

- 搜索算法
- 树
- 位运算

优化概述

对于给定的问题，当我们不易建立数学模型或者有数学模型但是解决起来很困难时，可以使用搜索实现。甚至可以说在信息学奥赛中，有一部分用其他算法的题目，也可以使用搜索来解决。但是搜索时间一般时间复杂度高，用来处理这些题目只能得到部分分，为了能得到更高的分数甚至满分，就要求我们学会更多算法或对搜索进行优化。常见的优化方法有以下两种：

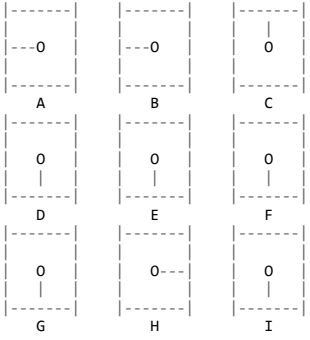
- 剪枝优化
- 位运算优化

位运算优化

- 位运算优化指的是将状态保存在二进制中，利用位运算的速度优势实现对状态查找、状态检查、状态转移的优化。
- 位运算优化没有改变搜索状态总数，所以位运算优化是“常数优化”。

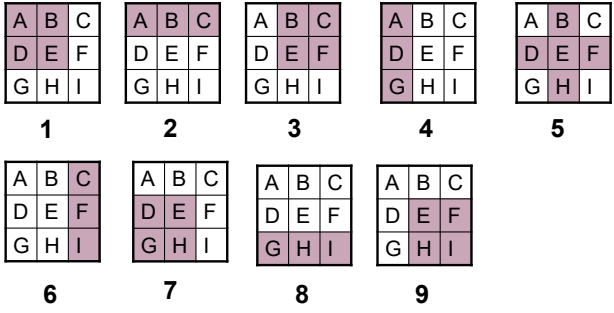
【例】时钟

考虑将如此安排在一个 3×3 行列中的九个时钟：目标要找一个最小的移动次序将所有的指针指向 12 点。表格中列出了 9 种不同的旋转指针的方法，每一种方法都叫一次移动。选择 $1 \sim 9$ 号移动方法，将会使在表格中对应的时钟的指针顺时针旋转 90 度。



移动方法	受影响的时钟
1	ABDE
2	ABC
3	BCEF
4	ADG
5	BDEFH
6	CFI
7	DEGH
8	GHI
9	EFHI

【例】时钟



例如，经过 $5 \to 8 \to 4 \to 9$ 的变换后，所有时钟都指向 12，但这不一定是答案。

```
9 9 12          9 12 12        9 12 12          12 12 12          12 12 12
6 6 6  /*5→*/   9  9  9 /*8→*/ 9  9  9 /*4→*/  12  9  9 /*9→*/  12 12 12
6 3 6           6  6  6        9  9  9          12  9  9          12 12 12
```

【例】时钟

【输入】
第 1-3 行: 三个空格分开的数字，每个数字表示一个时钟的初始时间，3,6,9,12。数字的含意和例子一样。

【输出】
单独的一行包括一个用空格分开的将所有指针指向 12:00 的最短移动顺序的列表。
如果有多种方案，输出那种使其连接起来数字最小的方案。(举例来说 5 2 4 6 < 9 3 1 1)。

【输入样例】

```
9 9 12
6 6 6
6 3 6
```

【输出样例】

```
4 5 8 9
```

【例】时钟

- 方案的顺序不重要，比如对于样例，采用 $5 \to 8 \to 4 \to 9$ 和 $4 \to 5 \to 8 \to 9$ 结果是一样的
- 每种方案最少使用 0 次，最多只能使用 3 次 (如果使用 4 次则全部时钟都回到最初状态，相当于没有进行过这种操作)。
- 总的状态数只有 ($4^9 = 262144$)，选择其中一个最小的输出即可。
- 可以使用搜索算法找每种方案使用的次数，一共 9 种方案，每种可取值 0∼3。

【例】时钟

```
1  int v[10], ans[10];    // 存储枚举步骤和答案步骤
2  int step = 0, min_step = 100;
3  // 当前搜索第 x 种移动方法的次数
4  void dfs(int x)
5  {
6      if(x == 10) {
7          bool ok = transform();  // 按照序列进行变换
8          if(ok && step < min_step) {
9              min_step = step;
10             for(int i = 1; i <= 9; ++i) ans[i] = v[i];  // 更新答案
11         }
12         return;
13     }
14     for(int i = 0; i <= 3; ++i) {
15         v[x] = i, step += i;   // 第 x 种变换进行了 i 次
16         dfs(x + 1);
17         v[x] = 0, step -= i;
18     }
19 }
20 // 主函数
21 dfs(1);
```

【例】时钟

- 上述方法中，由于变换方式没有规律，所以按照序列对时钟进行变换会花费大量时间。
- 时钟只有四种状态，12、3、6、9 点，分别用 0、1、2、3 表示；
- 用三位二进制来记录一个时钟的状态，它们对应的二进制数分别为：`000`，`001`，`010`，`011`；(为什么不用两位？)
- 需要将一个时钟指针旋转 90 度的时候，就给该时钟对应的二进制数加 1 (`001`)，再用按位与的方法去除高位 (即进位) 的 1，令最高的三位为时钟 A，最低的三位为时钟 I。
- 若想去除所有高位的进位 1，只需将 9 个时钟对应的二进制串按位与 `011011011011011011011011011(0x36db6db)` 即可。
- 每种旋转方案也有对应的二进制串，比如第 4 种方案“ADG”，对应 9 组二进制串 `001000000001000000001000000(0x1008040)`。

【例】时钟

- 例如，对于如下变换

```
9 9 12   3 3 0              0 3 0   12 9 12
6 6 6    2 2 2   /*--4-->*/  3 2 2   9 6 6
6 3 6    2 1 2               3 1 2   9 3 6
```

操作	二进制串
样例	01101100001001001001001010
4 号方案	00100000000100000000001000000
与 4 号方案相加	10001100001101001001100101 0
消除进位数字	01101101101101101101101101 1
消除进位	00001100001101001001100101 0

【例】时钟

```
unsigned int mov[10] = { 0, 0x1209000, 0x1240000, 0x241200,
    0x1008040, 0x209208, 0x40201, 0x9048, 0x49, 0x1209};  // 9 种移动方法对应的位
unsigned int carry = 0x36db6db;  // 去除进位
unsigned int st = 0;   // 时钟的初始状态

// 输入并形成初始状态
for(int i = 8; i >= 0; --i)
{
    int t; cin >> t;
    if (t == 12)     st = st | (0u << (i * 3));
    else if (t == 3) st = st | (1u << (i * 3));
    else if (t == 6) st = st | (2u << (i * 3));
    else if (t == 9) st = st | (3u << (i * 3));
}

bool transform()    // 根据变换序列 进行变换 判断能否到达目标
{
    unsigned int clock = st;
    for(int i = 1; i <= 9; ++i)
        for(int j = 1; j <= v[i]; ++j)
            clock = (clock + mov[i]) & carry;
    return !clock;
}
```

【例】N 皇后问题

N 皇后问题是由八皇后问题演变过来的，八皇后问题是国际西洋棋棋手马克斯·贝瑟尔于 1848 年提出：在 8×8 格的国际象棋上摆放八个皇后，使其不能互相攻击，即任意两个皇后都不能处于同一行、同一列或同一斜线上，问有多少种摆法。高斯认为有 76 种方案。1854 年在柏林的象棋杂志上不同的作者发表了 40 种不同的解，后来有人用图论的方法解出 92 种结果。我们将这个问题扩展为：在 $N \times N$ 格的棋盘上摆放 N 个皇后，使其不能互相攻击，即任意两个皇后都不能处于同一行、同一列或同一斜线上，有多少种摆法？

【例】N 皇后问题

【输入格式】
一行一个整数 $n(1 \leq n \leq 25)$。
【输出格式】
一行一个整数表示方案数。
【输入样例 1】

```
4
```

【输出样例 1】

```
2
```

【输入样例 2】

```
8
```

【输出样例 2】

```
92
```

【例】N 皇后问题

- 逐行尝试放置皇后，在每一行尝试每一个位置：如果可以继续下一行，如果不行尝试下一个位置，如果当前行都无法放置皇后则回溯。
- 设置三个标记数组c[],zd[],fd[]，来标记每一列、主对角线、副对角线上是否有皇后。

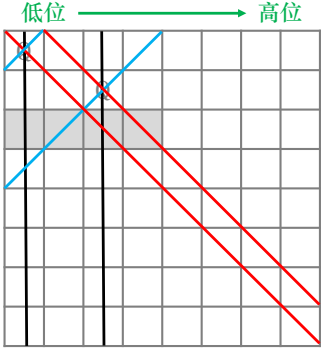
【例】N 皇后问题

```
1  const int N = 30;
2  int c[N], zd[2 * N], fd[2 * N]; // 列和对角线冲突数组
3  int g[N][N];                    // 棋盘
4
5  void dfs(int x) // 准备放置第 x 行的皇后
6  {
7      if(x == n + 1)  // 准备放置第 n+1 行的皇后，说明已经找到一个方案
8      {
9          ++ans;
10         print();
11         return;
12     }
13     for(int y = 1; y <= n; ++y)  // 枚举该行的每一个位置
14     {
15         if(c[y] || zd[x - y + n] || fd[x + y]) continue;   // 无法放置皇后
16         c[y] = zd[x - y + n] = fd[x + y] = true;  // 设置标记
17         g[x][y] = 1;     // 放置皇后
18         dfs(x + 1);      // 继续搜索下一行
19         g[x][y] = 0;     // 取消放置皇后，准备放在该行下一个位置
20         c[y] = zd[x - y + n] = fd[x + y] = false; // 释放标记
21     }
22 }
```

【例】N 皇后问题

- 上述算法需要尝试每行的每一个位置，需要判断位置是否可以放置皇后，这属于被动尝试。
- 思想修改为：每放置一个皇后，将它对下一行的影响传递下去，那么下一行放置皇后时就可以直接计算出可用位置集合，于是遍历可用位置放置皇后即可，对于不可能放置位置不用判断。

【例】N 皇后问题



放置第 i 个皇后受到已放置的前 $i-1$ 个皇后在列、主对角线和副对角线三个方面的约束。因此，我们定义 3 个位置状态变量：

1. c: 第 i 行哪些位置不能放皇后。例如左图对应的 c 为 `00000101` 。
2. zd: 在主对角线限制条件下第 i 行哪些位置不能放皇后。例如左图对应的 zd 为 `00001100` 。
3. fd: 在副对角线限制条件下第 i 行哪些位置不能放皇后。例如左图对应的 fd 为 `00000010` 。
4. 第 i 行不能放置皇后的位置的二进制为 `c | zd | fd = 00001111`
5. 第 i 行能放置皇后的位置的二进制 d 为 `~(c | zd | fd) & 11111111 = 11110000` 。[a]

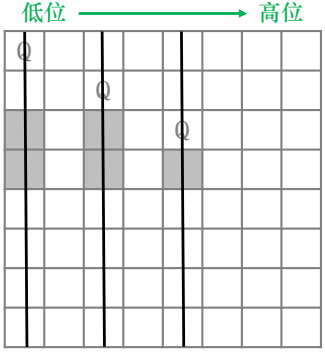
[a] 为什么要对全 1 进行按位与运算?

【例】N 皇后问题

- 放置位置选取：如何选择当前行可放置皇后位置 (从左到右)?
- 选择最左边的空位，即 d 的二进制中最低位的 1 的那一位`lowbit(d) = d & (-d)`。
- 目标状态：c 的二进制为全为 1，说明所有的皇后都已经全部放置。
- 如何设置目标状态?

```
1 target = (1 << n) - 1;
```

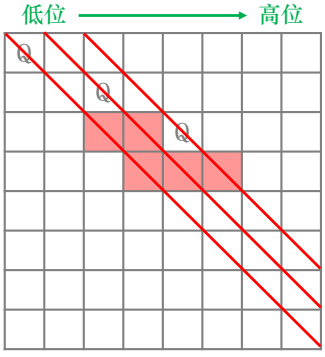
【例】N 皇后问题



在第 3 行第 5 列放置皇后对列标记 c 的影响：

- 原始 c= 00000101 。
- 可以放置皇后位置的二进制 d 为 11110000 ，于是放置皇后位置为lowbit(d)= 00010000 。
- 放置皇后之后 c = c | lowbit(d)= 00010101 。
- 放置皇后的位运算为：c = c | lowbit(d)。

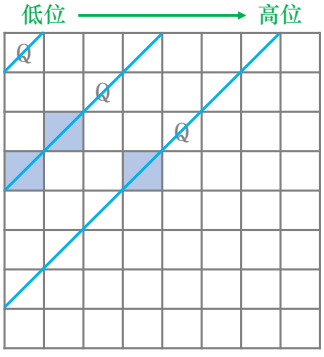
【例】N 皇后问题



在第 3 行第 5 列放置皇后对主对角线标记 zd 的影响：

- 原始 zd= 00001100 。
- 可以放置皇后位置的二进制 d 为 11110000 ，于是放置皇后位置为lowbit(d)= 00010000 。
- 放置皇后之后 zd = (zd | lowbit(d)) << 1 = 00111000 。
- 放置皇后的位运算为：
  zd = (zd + lowbit(d)) << 1。

【例】N 皇后问题



在第 3 行第 5 列放置皇后对主对角线标记 fd 的影响：

- 原始 fd= 00000010 。
- 可以放置皇后位置的二进制 d 为 11110000 ，于是放置皇后位置为lowbit(d)= 00010000 。
- 放置皇后之后 fd = (fd | lowbit(d)) >> 1= 00001001 。
- 放置皇后的位运算为：
  fd = (fd | lowbit(d)) >> 1

【例】N 皇后问题

```
const int N = 30;
int cnt, ans[N];
int n, target;
map<int, int> loc;  // 二进制位对应的皇后位置

void dfs(int x, int c, int zd, int fd)
{
    if(c == target)        // 找到一答案
    {
        ++cnt;
        return ;
    }
    int d = ~(c | zd | fd);
    d &= target;           // 将高位的 1 去除
    for(; d; d -= lowbit(d))
    {
        int p = lowbit(d);
        ans[x] = loc[p];       // 保存答案 p 的二进制中 1 所在的位置
        dfs(x + 1, c | p, (zd | p) << 1, (fd | p) >> 1);
        // 不用还原状态 因为递归后局部变量自动还原
        ans[x] = 0;
    }
}
```

【例】数独

数独是一种传统益智游戏，你需要把一个 9×9 的数独补充完整，使得图中每行、每列、每个 3×3 的九宫格内数字 $1 \sim 9$ 均恰好出现一次。

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

【例】数独

【输入格式】
输入共 9 行，对应于数独的行。每行给出一个正好为 9 位十进制数字的字符串，对应于此行中的单元格。如果单元格为空，它由 0 表示。
【输出格式】
程序应以与输入数据相同的格式打印解决方案。空单元格必须按照规则填充。如果解决方案不是唯一的，则程序可以打印其中任何一个。

【输入样例】

```
103000509
002109400
000704000
300502006
060000050
700803004
000401000
009205800
804000107
```

【输出样例】

```
143628579
572139468
986754231
391542786
468917352
725863914
237481695
619275843
854396127
```

【例】数独

- 该问题的搜索方案为：找到一个空白位置，填一个合法的数字，继续填下一个空白位置。
- 搜索的结果有两种：一种是所有的空白位置都填了合法的数字，说明找到了一个方案；一种是搜索过程中发现某一个空白位置没有合法数字可填，说明当前搜索失败，立即回溯。

```
1  // 准备填写第 k 个空白位置
2  bool dfs(int k)
3  {
4      if(k == m + 1) return true;  // 找到合法方案
5      for(能填写的数字)
6      {
7          填写数字;
8          if(dfs(k + 1)) return true;
9          恢复状态;
10     }
11     return false;
12 }
```

【例】数独

```
const int n = 9;
int g[n][n];
bool row[n][10], col[n][10]; // 行、列、格子出现的数字
bool grid[3][3][10];
int m = 0; // 空白区域的个数
struct Node { int x, y} a[100]; // 空白位置坐标

// 准备填写第 k 个空白位置
bool dfs(int k)
{
    if(k == m + 1) return true;
    int x = a[k].x, y = a[k].y;
    for(int i = 1; i <= n; ++i)
    {
        if(row[x][i] || col[y][i] || grid[x / 3][y / 3][i]) continue;
        row[x][i] = col[y][i] = grid[x / 3][y / 3][i] = true;
        g[x][y] = i;
        if(dfs(k + 1)) return true;
        row[x][i] = col[y][i] = grid[x / 3][y / 3][i] = false;
        g[x][y] = 0;
    }
    return false;
}
```

【例】数独

- 如果盲目搜索空白位置以及填写的数字，可能会导致搜索树靠近根的分支过多。
- 试想如果是人类玩数独，策略一定是先填上“已经能够唯一确定的位置”，然后从那些可填数字比较少的位置实施突破。所以每次选格子填数时，从所有空白位置里选择“能填的合法数字”最少的位置，考虑该位置上填什么数，作为搜索的分支，而不是任意选择位置填数。
- 在选择空白位置填数时，需要统计每一个空白位置可填数字的个数，如果使用前述标记需要大量检查行、列和九宫格标记的操作，可以考虑用位运算来进行标记和统计。

【例】数独

- 对于每行、每列、每个九宫格，分别用一个 9 位二进制数保存哪些数字已经填过。定义 $row[x]$ 表示第 x 行数字填写状态，定义 $col[y]$ 表示第 y 列数字填写状态，定义 $grid[][]$ 表示九宫格的数字填写状态。以样例中第 0 行第 1 列的位置为例：
  - `row[0]=100010101`
  - `col[1]=000100000`
  - `grid[0][0]=000000111`
- 对于位置 (x, y)，将 $row[x]$、$col[y]$ 和 $grid[x/3][y/3]$ 这 3 个二进制数做按位或（|）运算，就可以得到该位置已经填写的数字状态，然后与全 1 进行按位异或即可得到可以填写的数字状态。以样例中第 0 行第 1 列的位置为例：`(row[0] | col[1] | grid[0][0]) ^ 111111111 = 011001000`。

【例】数独

- 需要枚举可填写数字时，只需要对求出的上述标记状态 d 取 `lowbit(d)=d & (-d)`。以样例中第 0 行第 1 列的位置为例：`d=011001000`，那么可填的数字状态为 `lowbit(d)=000001000`，可填的数字为 4。
- 当一个位置填上某个数后，把该位置所在的行、列、九宫格记录的二进制数的对应位改为 1，即可更新当前状态；回溯时改回 0 即可还原现场。以样例中第 0 行第 1 列的位置为例，当填写数字 4 时：
  - `row[0] ^ 000001000=100010101 ^ 000001000=100011101`
  - `col[1] ^ 000001000=000100000 ^ 000001000=000101000`
  - `grid[0][0] ^ 000001000=000000111 ^ 000001000=000001111`

  回溯时的状态同理[1]。

[1]对二进制位按位异或（`^`）可以实现按需对位取反。

【例】数独

```
const int n = 9;
int g[n][n];
int m = 0;        // 空白位置总数
int row[n], col[n]; // 行、列、格子出现的数字
int grid[3][3];
int num[1 << n];  // 二进制中 1 的位置
int one[1 << n];  // 二进制数中 1 的个数

int lowbit(int x)
{
    return x & -x;
}

int one_cnt(int val)
{
    int res = 0;
    for (; val; val -= lowbit(val)) ++res;
    return res;
}
```

【例】数独

```
// 目前可填数字个数最小的位置
pair<int, int> can()
{
    pair<int, int> p = {-1, -1};
    int min_cnt = n + 1;
    for(int x = 0; x < n; ++x)
        for(int y = 0; y < n; ++y)
        {
            if(g[x][y]) continue;
            int d = row[x] | col[y] | grid[x / 3][y / 3];
            d ^= 0x1ff;
            if(one[d] == 0) return {-1, -1};  // 空白位置无法填数字直接回溯
            if(one[d] == 1) return {x, y}; // 只能填一个数字 直接填数字即可
            if(one[d] < min_cnt) p = {x, y}, min_cnt = one[d];
        }
    return p;
}
```

【例】数独

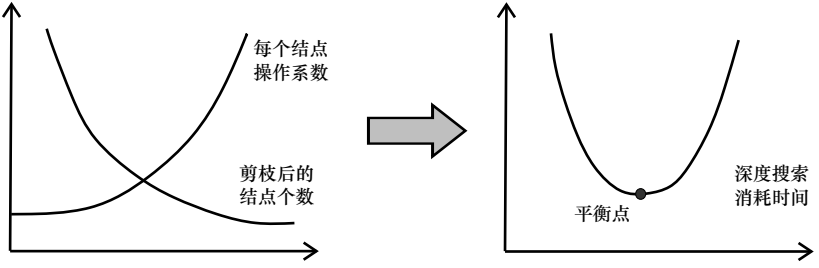
```
// 准备填写第 k 个空白位置
bool dfs(int k)
{
    if(k == m + 1) return true;
    pair<int, int> p = can();
    if(p.first == -1) return false;
    int x = p.first, y = p.second;
    int d = row[x] | col[y] | grid[x / 3][y / 3];  // 该位置已经填过的数
    d ^= 0x1ff;    // 该位置可以填的数字
    for(; d; d -= lowbit(d))
    {
        int t = lowbit(d);
        g[x][y] = num[t];
        row[x] ^= t; col[y] ^= t; grid[x / 3][y / 3] ^= t;
        if(dfs(k + 1)) return true;
        g[x][y] = 0;
        row[x] ^= t; col[y] ^= t; grid[x / 3][y / 3] ^= t;
    }
    return false;
}
```

练习

- 时钟 (COGS 668)
- 跳棋的挑战 (COGS 64)
- 数独 (COGS 3441/3444/3456)
- 靶形数独【NOIP 2009】(COGS 407)

总结

搜索消耗时间 ≈ 每个结点操作系数 × 结点个数



故，在优化时要综合考虑结点个数和结点操作系数，在可能的情况下：

- 减少结点个数——剪枝优化
- 减少每个结点的操作系数——位运算优化