

# 同濟大學

## 高级语言程序设计（基础）课程实验报告

### 孔明棋（*Peg Solitaire*）



学院 国豪书院

专业 人工智能(精英班)

姓名 余政希

学号 2452633

2025 年 5 月 28 日

目录

- 1. 题目背景与实践目标 ..... 3
  - 1.1. 题目背景 ..... 3
  - 1.2. 实践目标 ..... 3
- 2. 系统总体框架 ..... 3
  - 2.1. 系统架构 ..... 3
  - 2.2. 模块划分 ..... 4
    - 2.2.1. 主控模块 Game Controller ..... 4
    - 2.2.2. 界面模块 UI Manager ..... 4
    - 2.2.3. 棋盘模块 Peg Board ..... 4
    - 2.2.4. 求解器 Solver ..... 4
- 3. 界面与交互设计 ..... 5
  - 3.1. 主菜单设计与双缓冲技术 ..... 5
  - 3.2. 棋盘设计与径向渐变和镜面高光 ..... 5
    - 3.2.1. 径向渐变 ..... 5
    - 3.2.2. 镜面高光 ..... 6
  - 3.3. 提示功能设计与多线程 ..... 6
- 4. English Peg-Solitaire 33-Hole 最优求解器 ..... 7
  - 4.1. 求解器设计概述 ..... 7
  - 4.2. 状态编码与对称约化 ..... 7
    - 4.2.1. 位板编码 ..... 7
    - 4.2.2.  $D_4$  对称约化 ..... 8
  - 4.3. 一致启发式 ..... 8
    - 4.3.1. 三项评估指标 ..... 8
    - 4.3.2. 三路 Pattern Database (PDB) ..... 9
    - 4.3.3. 启发式合并策略 ..... 9
  - 4.4. IDA\* 搜索算法 ..... 9
    - 4.4.1. BF-layer-per-jump 预处理 ..... 9
    - 4.4.2. IDA\* 搜索 ..... 9
- 5. 功能展示与难点总结 ..... 9
  - 5.1. 功能清单 ..... 9
  - 5.2. 难点与解决方案 ..... 10
- 6. 结论与展望 ..... 10

# 1. 题目背景与实践目标

## 1.1. 题目背景

孔明棋也叫做法国独立钻石棋（Peg Solitaire），是一种智力游戏，通常由一个棋盘和若干颗棋子组成。

最经典的 English Solitaire 棋盘盘面如下图所示，一个 33 孔的盘面上摆放了 32 颗棋子。

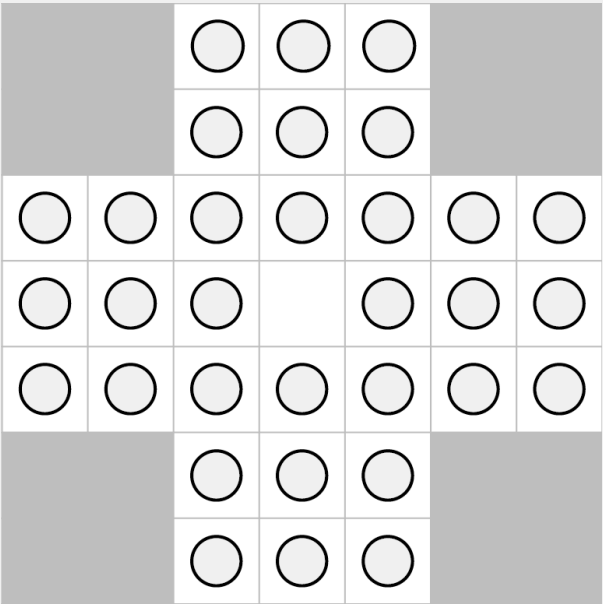


图 1.1.1: 经典英式 33 孔-孔明棋盘面

每次选定一颗棋子，跳过直接相邻的一颗棋子，并放到空的孔里，被跳过的棋子被移出棋盘。

每次移动可以是上下左右四个方向，但不能是斜向。

比如下图中的黑棋可以向左跳，但不能向上下或右侧跳。黑棋向左跳后，与他直接相邻的左边的棋子被移除。

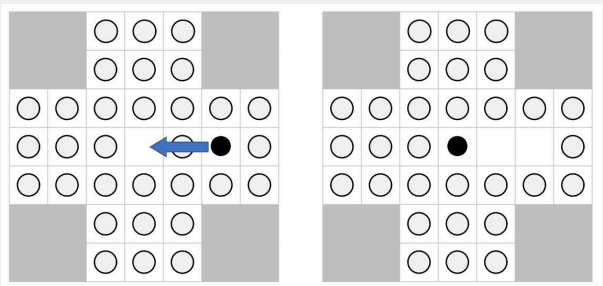


图 1.1.2: 孔明棋移动示例

如果盘面上只剩一颗棋子则胜利，如果剩下多颗棋子而且无棋子可移动则失败。

本项目旨在实现一个孔明棋游戏系统，提供人机交互界面和 AI 智能提示功能，帮助用户更好地理解和玩这个经典的智力游戏。

## 1.2. 实践目标

本实践完成了以下目标：

- 1. 使用 EasyX 绘制美观的图形化界面，提供使用鼠标的直观的人机交互。
- 2. 实现孔明棋的基本规则，包括棋盘初始化、棋子移动、胜负判断等。
- 3. 实现撤销、重做功能，允许用户在游戏过程中撤销之前的移动操作，或直接复原棋盘。
- 4. 实现提示功能，帮助用户找到最佳移动方案。

同时，在实现本项目的过程中，**学习和掌握**了以下技能：

- 1. 使用 C++ 和 EasyX 库进行图形化编程，绘制复杂的图形界面。
- 2. 理解和实现孔明棋的游戏规则和逻辑。
- 3. C++ 多线程编程，使用基础的多线程和原子变量基本实现 UI 和计算逻辑的分离。

# 2. 系统总体框架

## 2.1. 系统架构

本系统采用 C++ 与 EasyX 图形库实现，并在 Windows 操作系统下的 Visual Studio 平台开启 O2 优化编译通过，模块划分如下：

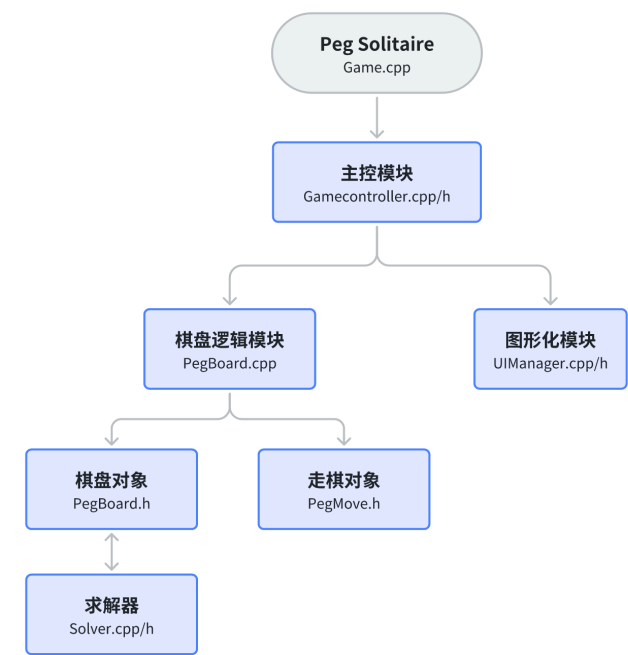


图 2.1.3: 系统架构图

## 2.2. 模块划分

### 2.2.1. 主控模块 Game Controller

负责游戏的整体控制逻辑，包括游戏初始化、棋盘状态管理、用户输入处理等。

通过 C++ 类实现：

```
1 class GameController {
2     PegBoard pegBoard; // 棋盘模块
3     UIManager ui; // 界面模块
4     std::stack<PegMove> history; // 记录走棋历史记录，便于
    撤销和重做操作
5 public:
6     void Run();
7 };
```

### 2.2.2. 界面模块 UI Manager

负责游戏界面的绘制和用户交互，包括棋盘、棋子、按钮等的显示和响应。

通过 C++ 类实现：

(由于代码较长，以下简化数值与传参)

```
1 class UIManager {
2 public:
3     static const int CELL_SIZE; // 单格像素
4     static const int MARGIN; // 边界留白
5     static const int BOARD_PIXEL;
6     UIManager(); // UI界面
7     void DrawBoard(); // 绘制棋盘
8     void DrawTips(); // 绘制底部提示信息
9     PegMove GetUserMove(); // 获取用户输入的移动
10    int DrawMenu(); // 绘制主菜单
11    void WaitForClick(); // 等待用户点击
12    void DrawButton(); // 绘制操作按钮
13    bool IsInButton(); // 判断鼠标是否在按钮区域
14 };
```

值得一提的是，为了实现求解提示功能，界面模块需要同求解器、棋盘对象进行交互。而为了统一管理和调用这些函数，界面模块需要通过主控模块交互来实现此操作。

因此在此处的 DrawBoard 函数与 GetUserMove 函数会从主控模块中获取求解器传递的最佳移动方案，并在界面上绘制提示信息。

### 2.2.3. 棋盘模块 Peg Board

负责棋盘的状态管理，包括棋子的布局、移动规则、胜负判断等。

通过 C++ 类实现：

(由于代码较长，以下简化数值与传参)

```
1 class PegBoard {
2 public:
3     static const int BOARD_SIZE = 7; // 棋盘大小
4     int board[BOARD_SIZE][BOARD_SIZE]; // 棋盘状态
5     PegBoard(); // 初始化棋盘
6     void Reset(); // 重置棋盘
7     bool CanMove(); // 判断移动是否合法
8     bool Move(); // 执行棋子移动
9     void Undo(); // 撤销上一步移动
10    bool HasValidMove(); // 判断是否有合法移动
11    int CountPegs(); // 统计棋子数量
12    PegMove GetBestMove(std::atomic<bool>& cancel);
13    // 获取最佳移动方案（与求解器交互）
14 };
```

值得一提的是，为了使求解器计算过程中能够实时刷新 UI 界面，笔者使用了多线程技术。

主控模块、棋盘模块通过原子变量与 UI 模块进行同步，确保在计算过程中 UI 界面能够正常响应用户输入。

PegMove 结构体则用来表示走棋的信息，包括起始位置、目标位置和被跳过的棋子。

```
1 struct PegMove {
2     int from_x, from_y;
3     int to_x, to_y;
4     int jumped_x, jumped_y;
5     PegMove():{} // 略，默认构造函数
6 };
```

### 2.2.4. 求解器 Solver

负责实现孔明棋的求解算法，提供最佳移动方案和提示功能。

为了实现棋盘与求解器的交互，此处接口需要提供求解器所需的棋盘状态信息，并返回最佳移动方案。

```
1 struct Jump { int from, over, to; };
2 // 用于表示一次跳跃的移动信息
3 void Solver_Init(); // 初始化求解器
4 bool solve(uint64_t start, std::atomic<bool>& cancel);
5 // 求解函数，同样也使用多线程
6 extern std::vector<Jump> bestPath; // 最佳路径
7 extern int xy2idx[7][7]; // 棋盘坐标到求解器索引的映射
8 extern int idx2x[]; // 求解器索引到棋盘 x 坐标的映射
9 extern int idx2y[]; // 求解器索引到棋盘 y 坐标的映射
```

具体的各模块实现细节将在下文中展开。

### 3. 界面与交互设计

#### 3.1. 主菜单设计与双缓冲技术

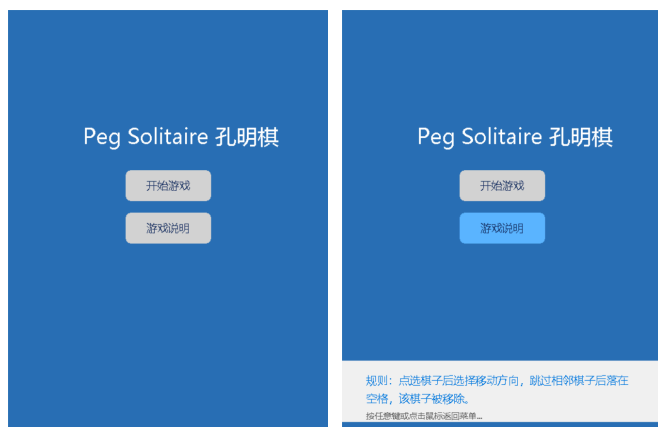


图 3.1.4: UI 界面设计：主菜单与规则

为实现主界面菜单，使用了 EasyX 图形库提供的各种文字和图形绘制函数。

而为了实现按钮悬浮提示且避免屏幕闪烁，实现更好的用户体验，笔者使用了**双缓冲技术**。

##### 为什么会屏幕高闪烁？

在使用 EasyX 绘制图形时，每次调用绘图函数都会直接在屏幕上进行绘制，这会导致屏幕频繁刷新，从而产生闪烁现象。

##### 双缓冲技术

为了解决屏幕闪烁问题，EasyX 提供了双缓冲技术。

BeginBatchDraw 和 EndBatchDraw 函数可以在屏幕上绘制一系列图形后，一次性刷新屏幕，避免了每次绘制都刷新屏幕导致的闪烁现象。

BeginBatchDraw 函数用于启用双缓冲，之后所有绘图操作都在内存中进行，直到调用 EndBatchDraw 函数才将结果一次性绘制到屏幕上。

FlushBatchDraw 函数用于刷新双缓冲区，将内存中的图形绘制到屏幕上。

EndBatchDraw 函数结束双缓冲，所有绘图操作都将被提交到屏幕上。

同时，鼠标并非一直移动，故 peekmessage 并不能实时获取鼠标位置，故需要新开一个变量记录鼠标位置，才能准确判断鼠标悬停状态。

下面是实现的关键代码。

```
1 int UIManager::DrawMenu() {
2     cleardevice();
3     ... // 绘制背景、标题等
4     int mx = -1, my = -1; //记录历史鼠标位置
```

```
5     ExMessage m;
6     BeginBatchDraw(); // 启用双缓冲
7     while (true) {
8         if (peekmessage(&m, EX_MOUSE)) {
9             mx = m.x;
10            my = m.y;
11            if (m.message == WM_LBUTTONDOWN) {
12                EndBatchDraw(); // 进入下一级，结束双缓冲
13                if (in_start_btn) return 1;
14                if (in_rule_btn) return 2;
15            }
16        }
17        cleardevice();
18        ... // 根据 mx,my 绘制按钮高亮状态
19        FlushBatchDraw(); // 刷新双缓冲
20        Sleep(10); // 避免 CPU 占用过高
21    }
22 }
```

#### 3.2. 棋盘设计与径向渐变和镜面高光

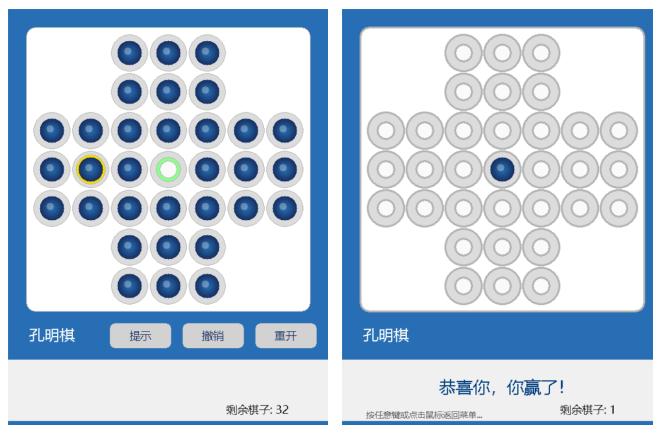


图 3.2.5: UI 界面设计：用户选棋提示与胜利界面

为实现美观的棋盘界面，笔者尝试使用了 EasyX 提供的各种绘图函数来试图实现棋子的径向渐变和高光效果。

##### 3.2.1. 径向渐变

为了实现棋子的径向渐变效果，笔者尝试使用了 fillcircle 函数来从大到小绘制深浅不一的圆形，试图模拟出渐变的效果。

但由于 EasyX 的神秘特性 (?)，该方法并不能实现真正的径向渐变，只会输出灰色且掺有蓝色斑点的圆形。

因此，笔者改用了手动计算每个像素点的颜色值，并使用 putpixel 函数绘制每个像素点的方式来实现径向渐变。

同时采用三通道非线性插值来更好地模拟渐变效果。

```
1 for (int dy = -RMAX; dy <= RMAX; ++dy) {
2     for (int dx = -RMAX; dx <= RMAX; ++dx) {
```

```

3      int dist2 = dx * dx + dy * dy;
4          // 计算当前点到圆心的距离平方
5      if (dist2 > R2) continue;
6      double t = sqrt((double)dist2) / RMAX;
7          // 非线性：让中间稍微突出、边缘过渡柔和
8      double fade = pow(t, 1.1);
9          // 三通道插值，模拟径向渐变
10     int r = GetRValue(cclrInner) + (GetRValue(cclrOuter) -
        GetRValue(cclrInner)) * fade;
11     int g = GetGValue(cclrInner) + (GetGValue(cclrOuter) -
        GetGValue(cclrInner)) * fade;
12     int b = GetBValue(cclrInner) + (GetBValue(cclrOuter) -
        GetBValue(cclrInner)) * fade;
13     putpixel(cx + dx, cy + dy, RGB(r, g, b));
14 }
15 }

```

### 3.2.2. 镜面高光

为了让棋子看起来更加立体，笔者添加了简单的镜面高光效果。

仿照径向渐变的实现方式，笔者计算了棋子左上角周围一定范围内的像素点，并对这些像素点进行颜色调整。

这里简单地将颜色值向白色偏移以模拟镜面反射效果。

关键代码如下：

```

1  int HR = RMAX / 3; // 高光半径
2  int hx = cx - HR / 2, hy = cy - HR / 2; // 高光中心点
3  int HR2 = HR * HR;
4  for (int dy = -HR; dy <= HR; ++dy) {
5      for (int dx = -HR; dx <= HR; ++dx) {
6          if (dx * dx + dy * dy > HR2) continue;
7          int x2 = hx + dx, y2 = hy + dy;
8          COLORREF base = getpixel(x2, y2);
9          // 简单地将颜色值向白色偏移，模拟镜面反射效果
10         int lr = min(255, GetRValue(base) + (255 -
            GetRValue(base)) * 0.3);
11         int lg = min(255, GetGValue(base) + (255 -
            GetGValue(base)) * 0.3);
12         int lb = min(255, GetBValue(base) + (255 -
            GetBValue(base)) * 0.3);
13         putpixel(x2, y2, RGB(lr, lg, lb));
14     }
15 }

```

## 3.3. 提示功能设计与多线程

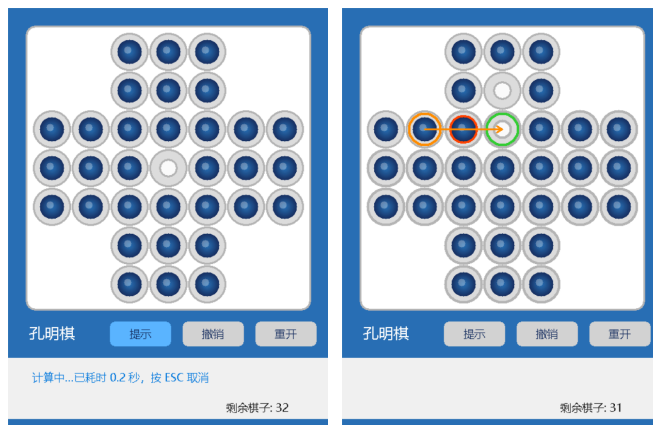


图 3.3.6: UI 界面设计：求解器求解与提示

为了实现在点击提示功能时能够实时刷新 UI 界面，笔者使用了 C++ 的多线程技术。

当用户在界面上点击提示按钮时，UI 模块会反馈给主控模块，主控模块会创建一个新的线程来执行求解器的计算。

这样，在求解器计算过程中，UI 界面仍然可以响应用户输入，并实时刷新显示。

同时，为了避免求解器计算时间过长或者用户手动停止导致线程阻塞，笔者使用了原子变量来控制线程的同步。

### C++ 多线程

C++11 标准库提供了 `std::thread`，可以非常方便地在同一进程中启动多个并发执行的“轻量级”线程。

```

1  std::thread worker([&](){
2      // 这里的 lambda 就是在新线程中执行的代码
3      result = pegBoard.GetBestMove(cancel);
4      done = true;
5  });

```

默认是“分离”线程，必须在主线程中调用 `join()`（等待线程结束）或 `detach()`（让线程后台运行）否则析构时会抛出异常。

### 线程捕获与数据共享

通过 `[&]` 按引用捕获外部变量 `result`、`cancel`、`done` 等，使得主线程和子线程可以对同一块内存进行读写。

### 线程阻塞与同步

`worker.join()`：在主线程最后调用 `worker.join()`，会阻塞（挂起）主线程，直到 `worker` 线程执行完毕才继续往下走。

```

1  worker.join(); // 阻塞等待子线程结束

```



忙等（Busy-Wait）循环：通过在主线程轮询做法，不断检查 done 变量的状态，同时更新 UI，直到子线程计算完成。

```
1 while (!done.load()) {
2     // 每 100ms 刷新一次 UI，并检测 ESC
3     Sleep(100);
4 }
```

当然，在更严谨的并发设计中，常用条件变量来替代忙等待，以避免 CPU 占用过高。

### 原子变量

当多个线程同时读写同一个普通变量时，会发生数据竞争（data race），导致未定义行为。

std::atomic 保证对该变量的读写操作都是原子性的，并且在没有额外锁的情况下也能在线程间及时地、安全地可见。

```
1 std::atomic<bool> done(false), cancel(false);
2 // 设置
3 done = true; // 等价于 done.store(true);
4 cancel = true; // 等价于 cancel.store(true);
5 // 读取
6 if (done.load()) { // 获取当前值
7     ...
8 }
```

这里简单完整展示了在此项目中的实现方式：

```
1 useAllHint = true;
2 AllHint = PegMove();
3
4 std::atomic<bool> done(false), cancel(false);
5 PegMove result;
6 auto t0 = std::chrono::steady_clock::now();
7 // 开始计时
8
9 std::thread worker([&]() {
10     result = pegBoard.GetBestMove(cancel);
11     done = true;
12 }); // 新建线程执行求解器计算
13
14 while (!done.load()) {
15     if (GetAsyncKeyState(VK_ESCAPE) & 0x8000)
16         cancel = true; // 用户按下 ESC 键取消计算
17     wchar_t buf[128];
18     double total = std::chrono::duration<double>(
19         std::chrono::steady_clock::now() - t0
20     ).count();
```

```
21     swprintf(buf, 128, L"计算中...已耗时 %.1f 秒，按 ESC 取消", total);
22     ui.DrawTips(buf, pegBoard.CountPegs(), true);
23     Sleep(100); // 每 100ms 刷新一次 UI
24 }
25
26 worker.join(); // 等待子线程结束
27
28 if (cancel.load()) {
29     ui.DrawTips(L"提示已取消", pegBoard.CountPegs(), true);
30 }
31 else {
32     AllHint = result;
33     wchar_t buf[128];
34     double total = std::chrono::duration<double>(
35         std::chrono::steady_clock::now() - t0
36     ).count();
37     swprintf(buf, 128, L"计算完成：用时 %.1f 秒", total);
38     ui.DrawTips(buf, pegBoard.CountPegs(), true);
39 }
```

## 4. English Peg-Solitaire 33-Hole 最优求解器

### 4.1. 求解器设计概述

本求解器追求「最少 moves」路径，其中连续使用同一棋子跳跃多次计为一次 move。

采用 BFSIDA\* 搜索算法，结合一致启发式函数和 Pattern Database（PDB）加速搜索。

同时，使用位板编码和  $D_4$  对称约化来减小搜索空间。

具体的，求解器设计分为以下几个部分：

### 4.2. 状态编码与对称约化

#### 4.2.1. 位板编码

使用 uint64\_t 的低 33 位表示每个孔位是否有棋子，实现状态压缩。

通过位运算实现对棋子状态的  $O(1)$  访问和修改。

```
1 using Board = uint64_t;
2 bool hasPeg(Board b, int idx) { return b >> idx & 1ULL; }
3 void setPeg(Board &b, int idx) { b |= 1ULL << idx; }
4 void clrPeg(Board &b, int idx) { b &= ~(1ULL << idx); }
5 int popcount(Board b) { return
    static_cast<int>(bitset<64>(b).count()); }
```

### 4.2.2. $D_4$ 对称约化

棋盘具正方形对称群  $D_4$  的八种对称，使用最小规范形  $\text{canon}(b)$  作为状态哈希，可将搜索空间缩减 8 倍以上。

具体地，在实现过程中，只需每次将八种对称均做出，并取字典序最小的状态即可。

关键代码如下：

```
1 Board canonical(Board b){
2   Board best = b;
3   for(int s=1;s<8;++s){
4     Board t = 0;
5     for(int j=0;j<HOLE_CNT;++j)
6       if(b >> fromSym[s][j] & 1ULL) t |= 1ULL<<j;
7     if(t < best) best = t; // 取字典序最小状态
8   }
9   return best;
10 }
```

同时，通过预处理  $\text{toSym}$  与  $\text{fromSym}$  映射矩阵，可在搜索过程中对称变换状态而不需重建整个棋盘，进一步减小常数。

```
1 void updateSymBoards(Node& child, const Node&
2   cur, const Jump& jp) {
3   child.sym = cur.sym;
4   for (int s=0;s<8;++s) { // 8 SYM
5     int pF = toSym[s][jp.from];
6     int pO = toSym[s][jp.over];
7     int pT = toSym[s][jp.to];
8
9     flipBit(child.sym[s], pF);
10    flipBit(child.sym[s], pO);
11    flipBit(child.sym[s], pT);
12  }
13  child.canon = min8(child.sym);
14 }
```

虽然  $D_4$  对称约化在减小搜索空间的同时，会增加状态转换的复杂度，但由于其对搜索空间的缩减，仍然是非常值得的。

事实上，在实际测试中， $D_4$  对称约化仅会增加不到 5% 的搜索时间，而记忆化存储空间与搜索空间却缩小了 8 倍。

## 4.3. 一致启发式

### 4.3.1. 三项评估指标

参考文献：Barker J, Korf R. Solving peg solitaire with bidirectional BFIDA[C], Proceedings of the AAAI Conference on Artificial Intelligence. 2012, 26(1): 420-426.

这里适当修改了原文中的启发式函数，增加了 Peg-Type 超量数和 Merson 区域占满数的势能考虑。

#### 角孔占据数 $h_c$

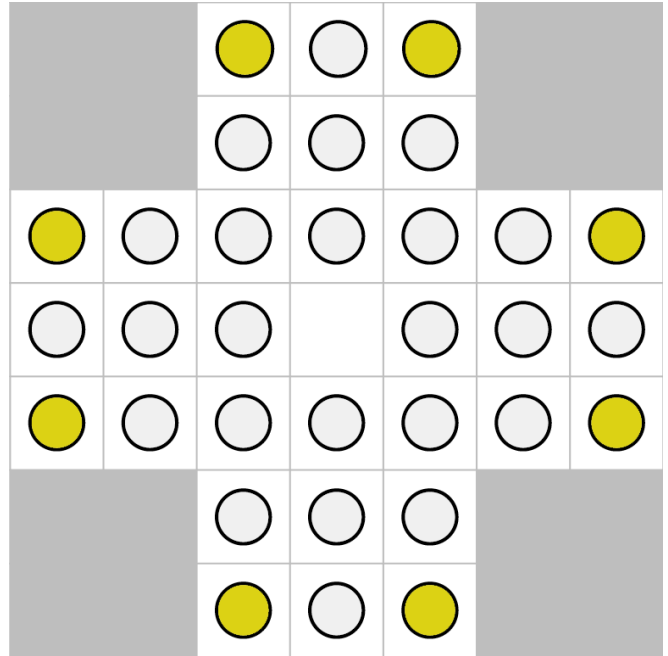


图 4.3.7: 角孔

如图，黄色的孔即为孔明棋的角孔（corner holes），是指棋盘角上的孔位。

这些孔位无法通过其他孔位跳跃到达，只能通过本身向外跳跃才能消去。

利用该性质，我们定义  $h_c$  为棋盘上角孔的占据数。

#### Peg-Type 超量数 $h_t$

注意到，将孔明棋孔位根据所在行列的奇偶性分为四类，显然跳跃并不能“跨类”。

下图不同颜色的孔即展示了同一类孔位，通过势能分析，容易得在理想状态下，对于每个 Peg-Type，单次 move 可以消去 4 个同类型的棋子。

但显然这是不足以支持所有棋子消去的，因此在实际操作中，我们对四种 Peg-Type 的棋子的势取 max 而非 sum。

由于本文篇幅有限，故不再展开具体的证明与分析。

利用该性质，我们定义  $h_t$  为各 Peg-Type 超量数  $\lceil \frac{x}{4} \rceil$  最大值。

#### Merson 区域占满数 $h_m$

原论文中的 Merson 区域定义在 Wiegleb 盘面上，（除去第一类角孔后），若每个区域中均填满棋子，则区域中的棋子仅能被来自该区域中的棋子跳跃消去。



然而在英式孔明棋中，棋盘的形状与 Wiegleb 盘面不同，且难以高效地定义常规的 Merson 区域。

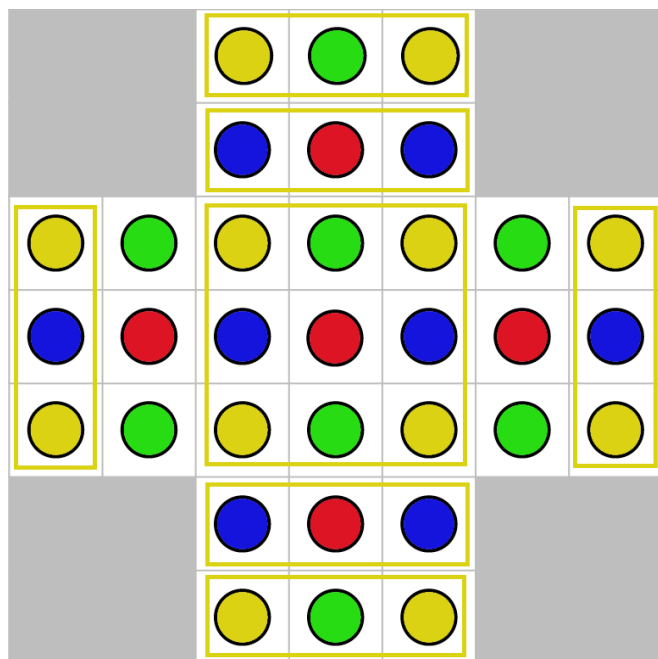


图 4.3.8: Peg-Type 与 Merson

注意到，如果将棋盘按照上图分为 7 个区域（且除去第一类角孔后），若区域中均填满棋子，则区域中的棋子**不一定**仅能被来自该区域中的棋子跳跃消灭。

但在理想情况下，不符合上述结论的情况少之又少。由于本文篇幅有限，故不再展开具体的证明与分析。

因此，笔者在此处仍认为其具有 Merson 区域的性质。

利用该性质，我们定义  $h_m$  为 Merson 区域占满数。

#### 4.3.2. 三路 Pattern Database (PDB)

将棋盘划分为三个不重叠区域（左臂、中央十字、右臂），各含 11 个孔。

```
1 int L_idx[11] = { 0, 1, 2, 3, 6, 7, 8, 9, 12, 13, 14 };
2 int C_idx[11] = { 15, 16, 17, 18, 19, 22, 23, 24, 25, 26, 27 };
3 int R_idx[11] = { 28, 29, 30, 31, 32, 4, 5, 10, 11, 20, 21 };
cpp
```

将整个问题空间划分为若干互不干扰的子空间，每个子空间建立一个数据库，记录从任意子状态到目标状态的最少步数。

搜索时将当前状态“压缩”为多个子状态，在 PDB 中查表，多个启发式相加即可作为整个局面的启发式评估值。

构建启发式： $h_{p(b)} = \sum_{k=1}^3 p_{k[C(b,k)]}$

每个子图反向 BFS 构建数据库（共  $3 \times 2048$  项，约 6 KB），记录从该子图到仅剩一子的最少跳跃数。由于子图间无交叉，和仍具一致性。

#### 4.3.3. 启发式合并策略

在实际应用中，对两种启发式合并策略取 max 作为  $h$  值，弥补各启发式覆盖盲区。

### 4.4. IDA\* 搜索算法

#### 4.4.1. BF-layer-per-jump 预处理

区别于传统 move-per-layer 模式（每步 move 抽象为一系列 jumps），本系统采用 jump-per-layer 模式，每层仅考虑一步 move，因此不再需二次搜索生成 move 集合，直接遍历静态 vector 即可。

同时，移除 pegs 只需要记录历史跳跃信息即可，避免了重复计算。

跳跃表共 76 条，提前离线生成，仅遍历即可扩展所有后继。

#### 4.4.2. IDA\* 搜索

初始阈值  $\text{thres} = h(\text{start})$ ，每轮根据 DFS 中的 nextThreshold 更新。

每轮搜索以固定上限  $\text{thres}$  展开 DFS。采用如下代价计量方式：

- 若连续使用同一棋子跳跃，则 move 代价  $g$  不变；
- 否则  $g$  增加 1；
- $f = g + h(b)$ ；
- 若某状态规范形已访问且已有更优  $g$  值，则剪枝；
- 每轮若无解，则提升  $\text{thres}$  为本轮  $f$  超过限制的最小值。

由于  $h$  为一致启发式（monotone），故  $f$  单调不减，满足 IDA\* 正确性要求。

注：理论上也能改为双向 BFSIDA\* 搜索，但还需注意孔明棋在正向搜索时，因其起点状态为中央为空，故实际有效状态约 2%，但当反向搜索时，会抵达中央非空的状态，因此反向搜索效率较低。

测试环境：Core Ultra 7 155H，编译平台：Windows 11 24H2。

在开启 O2 优化的随机情况下，且初始状态困难度较高，该求解器仍然能在 5s 内找到最优解。

代码部分较为复杂，本文篇幅有限，故不再展开，感兴趣的读者可参考完整代码。

## 5. 功能展示与难点总结

### 5.1. 功能清单

1. **棋盘可视化**：可视化绘制棋子和空位
2. **用户交互**：鼠标点击完成选中与跳跃

3. **智能提示**：实时计算推荐跳法，显示路径
4. **计算进度提示**：显示耗时、取消中断机制
5. **撤销与重做**：支持撤销上一步或重置棋盘
6. **重置与退出**：ESC/Enter 支持控制流程

具体的，推荐在 Visual Studio 中打开本项目，开启 O2 优化，选择 Release 编译并运行，实际体验项目的魅力。

## 5.2. 难点与解决方案

在本项目中，笔者遇到了以下几个难点，并通过相应的解决方案克服了这些难点：

1. **图形界面绘制**：使用 EasyX 绘制复杂的图形界面，尤其是棋盘和棋子的渐变效果。通过手动计算每个像素点的颜色值，实现了径向渐变和镜面高光效果。
2. **多线程与 UI 刷新**：在求解器计算过程中，如何保持 UI 界面的实时刷新。通过 C++ 的多线程技术和原子变量，实现了 UI 界面与求解器的分离，使得 UI 界面能够在求解器计算时仍然响应用户输入。
3. **一致启发式与 PDB**：如何设计一致的启发式函数，并构建 Pattern Database (PDB) 来加速搜索。通过将棋盘划分为三个不重叠区域，使用反向 BFS 构建数据库，实现了高效的启发式评估。
4. **状态编码与对称约化**：如何使用位板编码和  $D_4$  对称约化来减小搜索空间。通过位运算实现对棋子状态的  $O(1)$  访问和修改，并使用最小规范形作为状态哈希，减少了搜索空间。
5. **求解器设计**：如何设计一个高效的求解器，能够在合理时间内找到最优解。通过 IDA\* 搜索算法和一致启发式函数，结合多线程技术，实现了高效的求解器。

## 6. 结论与展望

本项目成功实现了一个孔明棋游戏系统，提供了美观的图形界面和智能提示功能。

通过使用 C++ 和 EasyX 图形库，笔者不仅实现了孔明棋的基本规则和人机交互界面，还学习探索了多线程编程、图形绘制、启发式搜索等技术。

未来可以在以下几个方面进行改进和扩展：

1. **优化求解器算法**：进一步优化求解器的算法，提高计算效率，尤其是在初始状态较复杂时。
2. **增加更多游戏模式**：除了经典的孔明棋模式外，可以增加更多变种或自定义棋盘布局，增加游戏的趣味性。
3. **增强用户体验**：改进 UI 界面设计，增加音效和动画效果，使游戏更加生动有趣。
4. **跨平台支持**：将项目移植到其他操作系统或平台，如 Linux 或 macOS，增加用户群体。
5. **AI 对战模式**：实现 AI 对战模式，让用户可以与 AI 进行对战，增加游戏的挑战性。

由于笔者正处于期末月，时间有限，还需完成其他汇报答辩合计四项、综述长文两篇、小作文一篇、期末考试

四门，故本文仅为初步实现，未能实现更复杂的功能，若在后续能有时间进一步实现，将会更新。

但笔者仍然非常荣幸能在四天的时间内完成了这个项目，并在此过程中学习和掌握了许多新的技能。

同时，本文之所以能够面世，离不开龚老师的指点与丰富的互联网资源的帮助。

感谢您的阅读，期待您的反馈和建议！