程序设计II大作业实验报告

姓名：刘良宇

学号：PB20000180

实验题目与要求

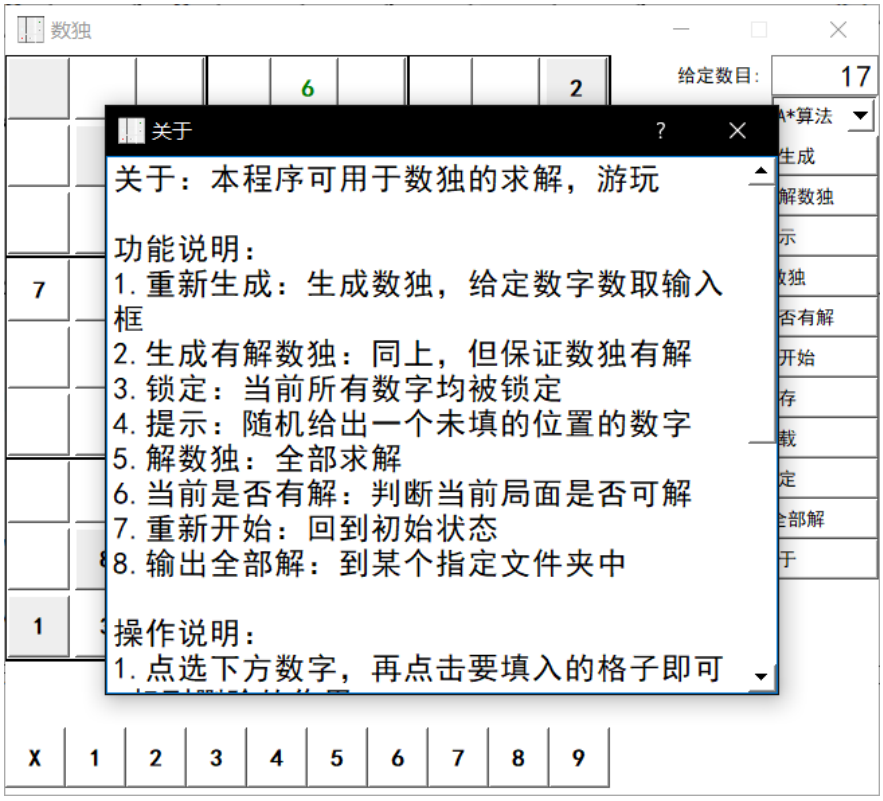
本次实验主要内容是实现一个简单的数独软件，具体要求如下：

基本功能部分

程序说明信息以及交互

程序的基本交互由图形化界面实现。

说明信息见 “关于” 功能按钮。



打印数独

通过图形化界面表示。

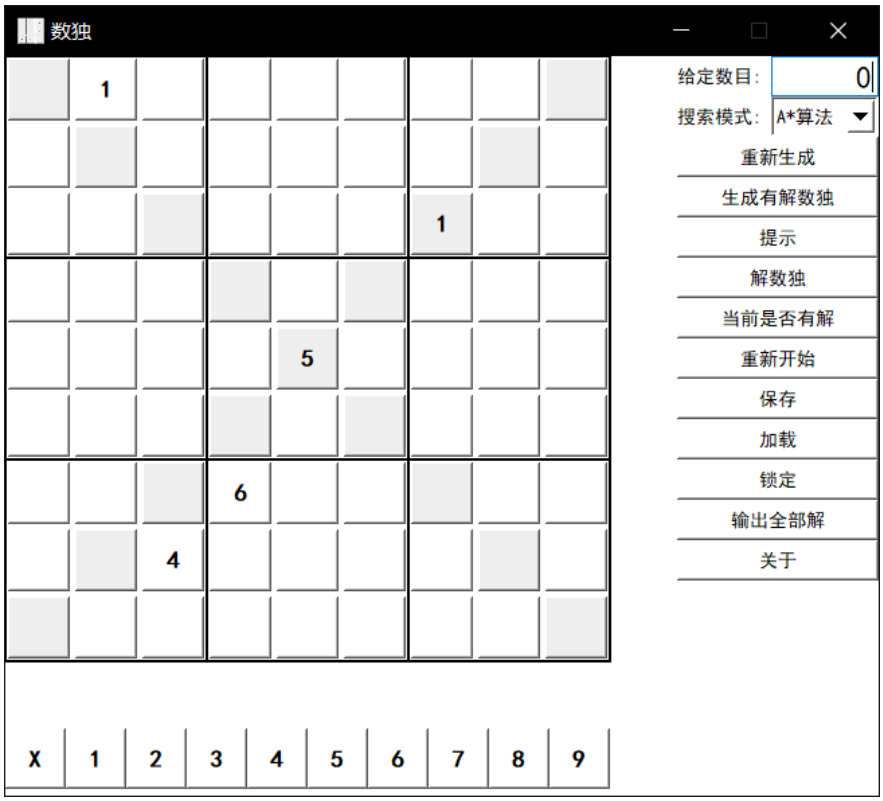
黑色表明是初始状态（被锁定，玩家无法修改），蓝色是玩家自行填入的数字，绿色是当前提示的数字（如果再进行提示等操作，则会变为蓝色）。

输入数独

初始化一个数独的方式有两种：

支持玩家生成一个大小为0的空白数独

之后可以通过底端操作区域填入数字。填入数字完成后，点击锁定，即可初始化。填入数字过程如果矛盾会有提示。（详见后面错误处理部分）。



可以通过读档的方式完成

详见后面读档部分。

生成数独

程序的第一个输入框可以输入指定数目的数字，指定初始生成的数独带有多少数字。如果数字范围异常（0到40之外），则会报错提示。

输出数独的解

点击解数独按钮，即可获得当前局面的解。如果无解，本程序会提示。

填入数字合法情况

采用下列方式向数独中填入数字

1. 点击下方9个候选数字或代表删除的“X”。
2. 点击要填入或删除数字的格子

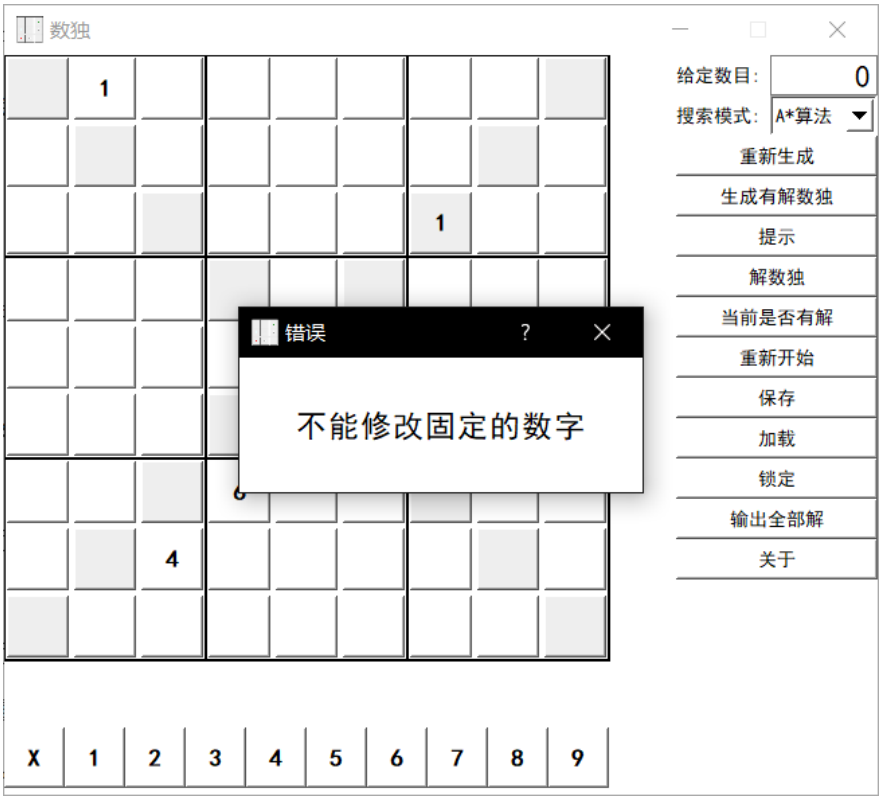
对于合法输入，填入/删除数字会直接反馈在图形界面上。

填入数字异常情况

处理以下异常输入：

- 如果试图修改题目固定的数字

  此时，会出现以下提示：



- 如果填入的数字过大

  这一情况不会发生。图形界面仅允许输入1-9的数字或者删除数字。

- 如果输入的填入位置不合法

  这一情况不会发生。图形界面仅允许填入该81个格子。

- 如果填入的数字和已有数字重复

  此时会提示具体重复的行列。

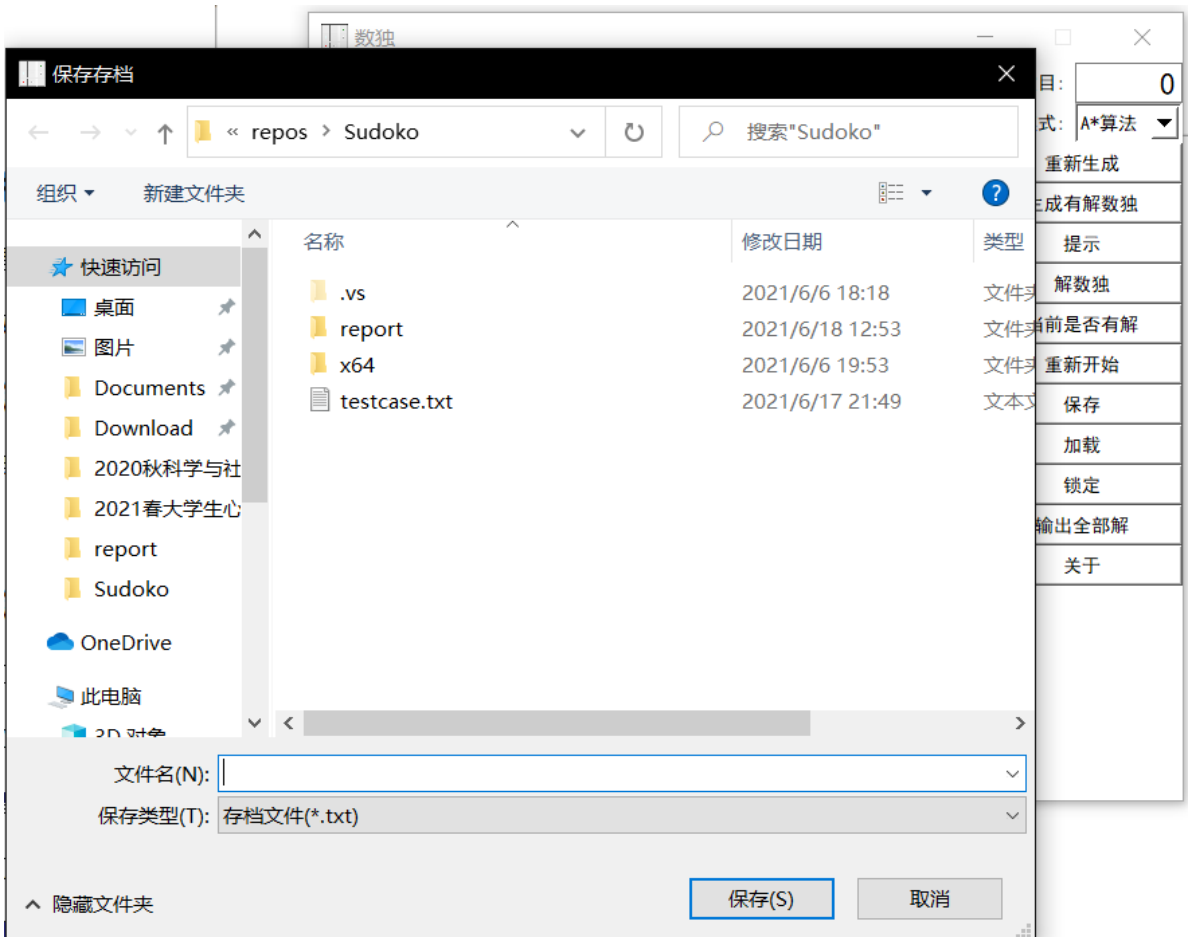
数独提示信息

点击提示按钮。此时，如果数独有解，会检查是否存在逻辑上一步可以推理出唯一解的格子，如果有，则填入，否则填入可填入可能性最少的格子。

如果数独无解，则会给出无解提示。

保存游戏状态

点击保存。会调用Windows提供的文件选择界面。指定文件夹，填入文件名后即可保存文件状态。



加载游戏状态

可以加载存档文件。效果同上。

存档文件格式：

首先是一个9*9的矩阵，记录了数独每个位置上的数字。没有放置计为0。

其次的9*9矩阵，如果为0代表这一位不是初始的数字，如果不为0则代表这一位是初始的数字。

所以可以通过输入两遍相同的矩阵，通过读档功能完成数独的初始化。

扩展部分

图形化界面

已经通过qt实现。

高级搜索技巧

实现了迭代加深dfs以及A*搜索。

具体介绍见下文。

搜索模式可以通过下拉栏切换。

多线程优化

实现使用多线程技术并行搜索输出数独全部的解。保存入指定文件中。

仅保存前100万个搜索到的解，以避免文件过大。

100万这个数字通过宏定义，可以方便地修改。

具体设计

该部分将**简单介绍**实验中的函数拆分与图形化实现的细节，例如，包括哪些函数，函数参数是什么，作用是什么等。扩展功能的介绍等。

图形化实现

通过qt实现。main.cpp分析：

```
int main(int argc, char *argv[])
{
    //防止文件名中文乱码
    QTextCodec* codec = QTextCodec::codecForName("UTF-8");
    QTextCodec::setCodecForLocale(codec);

    //指定应用程序
    QApplication a(argc, argv);

    //全局字体设置
    QFont font;
    font.setPointSize(14);
    font.setFamily("黑体");
    a.setFont(font);

    //主要窗口对象
    Sudoku w;
    w.show();

    //让应用程序对象进入消息循环（代码阻塞）
    return a.exec();
}
```

主要是生成了窗口对象w。这是主窗口。属于Sudoku类。

随后`return a.exec()`避免了代码中止，相当于让程序进入死循环，点击关闭按钮才会关闭。

而主要的控件位于Sudoko.h中：

```
//控件
QLineEdit* input_num_given; //输入框（输入生成数独时给定数字）
QComboBox* combox_search;   //搜索模式选择
QPushButton* btn[11];   //操作按钮
QSudokuBtn* sudoku_btn[9][9];   //9*9数独按钮
QSudokuBtn* sudoku_input[10];   //数独输入按钮(0-9，0代表归零)
```

包含函数

```
//槽函数
void search_mode_change();  //更改搜索模式
void regenerate();    //生成数独，不保证有解，并绘制
void genrt_with_solu(); //生成数独，保证有解，并绘制
void replay();  //回到初始状态
void lock();    //固定当前数独
void solve();   //解数独
void hint();    //提示
void if_solu(); //是否有解
void about();   //关于
void sudoku_click(int x, int y);    //填入数独
void save();  //保存当前数独
void load();  //加载数独
void all_solu();    //显示所有解答

//图形化辅助函数
void initial();    //初始界面创建
void create_widget(char* title, char* content); //创建错误/提示窗口
void sudoku_paint();    //数独数字全部更新

//计算函数
int search();  //搜索函数，无解时返回0，有解时确保sudoku_solve中是解
int search_dfs(int cell);  //朴素dfs，无解时返回0
int search_dfs_ID(SudokuNode node_now, int depth, bool full_search);  //迭代加深
dfs
int* search_dfs_get_tree(SudokuNode node_now, int depth);  //获得树的结构
void search_dfs_all(SudokuNode node_now, int depth, int recur_depth, bool
if_parent);  //dfs输出全部解
void output_to_file(SudokuNode& node);  //输出到文件
int search_astar();    //A*搜索算法
void generate(int size);    //生成数独，不保证有解

//错误处理
int if_repeat_error(int i, int j, int x);    //是否有数字重复错误
void repeat_error(int i, int j);    //i行j列数字重复，报错
```

为了方便说明，这里再列出本程序的一些基本数据：

```
char sudoku_num[9][9] = { 0 };  //数独当前状态数字
char sudoku_solu[9][9] = { 0 };    //数独解
bool fixed[9][9] = { 0 };   //是否被固定（是否是初始状态）
```

1. 槽函数对应了各个控件，在控件发生对应的事件时（本程序中，除了更改搜索模式，均是点击按钮操作），会触发槽函数。槽函数和触发它的信号之间通过connect函数连接。

2. 图形化辅助函数中，`initial()`用于建立初始界面，包含了控件位置，大小的处理以及信号和槽函数的连接。

   `create_widget()`用于创建窗口，两个参数指定创建窗口的标题和内容。通常用于处理各种报错。

   `sudoku_paint()`的调用是在解完数独后，需要更新所有数独格子的显示。

3. 计算函数中，`search()`是搜索函数。返回值1代表有解，并且解会被储存在sudoko_solu中，返回值0代表无解。这个函数会依据当前的搜索模式，调用不同的函数完成搜索任务。也就对应了`search_dfs(),search_dfs_ID(),search_astar()`这三个函数，也就是普通dfs，迭代加深dfs和Astar搜索。这三个搜索函数同样返回1代表有解，0代表无解。

   `search_dfs_get_tree(),search_dfs_all(),output_to_file`功能上是并行输出所有的解，与以上搜索函数略有不同。

   `generate()`作用是随机生成指定个数字的数独（但是不保证有解，仅保证不出现矛盾）存放到sudoku_num中，并更新fixed数组。

4. 错误处理函数主要是重复的处理。需要单独判断是哪一行哪一列出现了重复。

拓展功能——迭代加深dfs

在介绍 *高级搜索方法* 之前，先对搜索数独的过程做一个抽象：取每个局面作为一个结点。

```
class SudokuNode    //数独的一个局面
{
public:
    int cost_so_far, cost;  //A*算法使用
    char num[81];   //当前局面各个位置的数字
    bool mark[81][10];  //标记每个位置能放什么数字，false代表可以放
    char num_can_put[81];   //标记每个位置能放多少个数字
    //运算符重载（用于A*中使用stl的优先队列）
    friend bool operator<(SudokuNode n1, SudokuNode n2) {
        return n1.cost > n2.cost;
    }
    //构造函数
    SudokuNode();
    SudokuNode(char src[][9]);
    //运算函数
    static void mark_cell(int cell, bool* mark, char* num); //标记cell可放数字
    int fill(int cell, int num_fill);    //填数，返回0代表填入后会无解
    int dis_uni();  //填入所有有唯一解的格子
    void cal_cost(int increase);    //A*搜索中计算cost总
    void get_current_num(char to_num[][9]); //将当前结点信息输出
};
```

接下来介绍迭代加深算法的实现。对于一个简单的dfs，会从第一个格子开始顺序深度优先遍历，这样一来遍历的层数一定是81层。

但这样dfs的效率过于低下，所以首先考虑每次选出可填入数字选择最少的格子，然后对它遍历。

此外，在搜索过程中，如果某个格子有唯一解，可以直接填入。

经历了以上的改编过后，dfs的效率大大提高。也可以实现迭代加深的改编。

IDDFS搜索函数如下所示：

```
int Sudoku::search_dfs_ID(SudokuNode node_now, int depth, bool full_search) {
```

```
    if (depth > 24 && !full_search) {
        return 0;   //迭代加深，退出递归。
    }
    SudokuNode node_new;

    int dis_uni = node_now.dis_uni();   //填入只有唯一解的格子
    if (dis_uni < 0)return 0;   //无解，返回0
    if (dis_uni == 0) {
        node_now.get_current_num(sudoku_solu);      //有解，输出
        return 1;
    }
    dis_uni--;  //此时dis_uni是可填入数字可能最少的格子，范围0到80

    for (int j = 1; j <= 9; j++) {  //填入可能最少的格子进入下一层
        if (!node_now.mark[dis_uni][j]) {
            node_new = node_now;
            if (!node_new.fill(dis_uni, j)) {
                continue;   //如果填入后出现无解，跳过
            }
            if (search_dfs_ID(node_new, depth + 1, full_search)) {
                return 1;
            }
        }
    }
    return 0;
}
```

`bool full_search`指定了需不需要搜索全部的层数。经过实际实验记录，第一次只搜24层是一个合理的选择。

```
if (search_dfs_ID(start_node, 0, false) || search_dfs_ID(start_node, 0, true))
```

就可以用于判断当前局面是否有解。

拓展功能——Astar搜索

同上，认为数独的每个状态都是一个结点，那么这些结点具有以下几个特点：

1. 单向性：永远是填入n个数字的指向填入$n+1$个数字的结点。
2. 层次性：由填入数字的个数可以对结点进行分层。
3. 每个结点所连接的结点数目是很大的。

对于第三条特别考虑，这意味着算法实现时必须考虑剪枝，否则占用的时间，空间都会很大。而正因为数独具有第二条所示的层次性，我们做下列考虑：

假设数独有解，初始给定n个数字，那么对于一个可行的解法，考虑数字放入先后顺序，一定对应一条从第n层一直到第81层的路径，但如果不考虑数字放入先后顺序，那么一共应该有$(81-n)!$条路径。我们只关心数独的解，所以希望将这些路径归一。在数学上，这有个很好的解释：假设我们每一层都指定一个格子，这一层只能在这个格子放数字，那么刚好就指定了一条特定的路径，且这样做不重复不遗漏。不妨每一层都选择当前可填入数字最少的格子填数字，则这样可以最好的减少计算资源的消耗。

有了这个想法之后，我们来选择$f(n)=g(n)+h(n)$。在这个案例中，代价可以作为计算资源的代价理解，而不是结点间距离，毕竟每一层结点间距离都是1，不好作为判断依据。可以取每次选择的选择最少的结点的选择数，也就是从一个结点出发，能到达下一层的几个结点，这衡量了计算资源的消耗。

那么$g(n)$的选择已经呼之欲出了。对于初始结点，取$g(n)=0$，对于结点P，如果选择填入数字的格子有k个选择，那么对由这个选择到达的新的结点Q，$g(Q)=g(P)+k$。

但是$h(n)$的选择我们只能近似做一个估计。这里给出作者的一个估计式：设当前局面中，未填入数字的格子为$a_1, a_2, \ldots, a_n$，每个格子有$b_1, b_2, \ldots, b_n$种填入数字的选择，可以考虑取$h(n) = k\sum_{1}^{n} b_n$，其中k是一个待定的常数。$\sum_{1}^{n} b_n$反映了场面还有多少种可能性，这个值越小，可以认为离目标（终点结点）的距离越近。那么该如何选取k呢？一方面，考虑$h(n), g(n)$的换算关系，对于一个初始的空局面，$g(n) = 0 + 9$，增加了9，而在填入数字之后，$\sum_{1}^{n} b_n$减少了26，这样一看，k在3附近比较合理。另一方面，可以在写出搜索函数后实际进行试验，初始的$h(n)$应该比末状态的$g(n)$略小，也就是实际的最短路径长度会大于一开始的预估距离。

经过综合考虑，$k = 4$是一个合理的取值。搜索代码如下所示：

```
int Sudoku::search_astar() {
    SudokuNode current(sudoku_num), next;
    std::priority_queue<SudokuNode> frontier;
    frontier.push(current);
    while (!frontier.empty()) {
        current = frontier.top();
        frontier.pop();

        int min_index = current.dis_uni();  //寻找最少可能的格子
        if (min_index < 0)continue;
        if (min_index == 0) {
            current.get_current_num(sudoku_solu);
            return 1;
        }
        min_index--;

        //对这个格子生成所有可能的新节点，并加入优先队列
        int cost_now = 0;
        for (int j = 1; j <= 9; j++) {
            if (!current.mark[min_index][j]) {
                cost_now++;     //计数当前有多少选择
            }
        }
        for (int j = 1; j <= 9; j++) {
            if (!current.mark[min_index][j]) {
                next = current;
                if (next.fill(min_index, j)) {
                    next.cal_cost(cost_now);
                    frontier.push(next);
                }
            }
        }
    }
    return 0;
}
```

拓展功能——多线程搜索

`search_dfs_get_tree()`,`search_dfs_all()`,`output_to_file()`是需要用到的函数。

相关变量：

```
//多线程相关
int res_num = 0;    //一共有多少个解
int new_thread = 0; //当前子线程数
std::thread* next[MAX_THREAD];  //存储子线程
const int max_thread = MAX_THREAD; //最大线程数
std::mutex mut_res_num, mut_out;    //互斥锁 确保线程安全
QTextStream* file_out;   //文件输出流
```

`MAX_THREAD`通过宏定义。所有线程共享`QTextStream* file_out`这一文件输出流，需要互斥锁`mut_out`。为了获得总共的解数量，还需要`mut_res_num`。

多线程的基本实现思路是基于上文所说的优化后的dfs算法，对于一个递归进行的算法而言，重要的是将它拆分成运算量相当的一些子任务，每个任务需要消耗一个线程。对此，采用`search_dfs_get_tree()`函数获得前四层各对应多少个子结点，然后选择合适层数（恰好小于指定的最大线程数），在该层，`search_dfs_all()`会分离出新的线程。

为了等所有线程结束后再输出解的数量，需要暂存这些线程，使得可以通过`join()`方法堵塞函数。

`output_to_file()`则调用文件输出流，输出当前的解。

相关函数如下所示：

```
void Sudoku::all_solu() {
    ...
    int* p = search_dfs_get_tree(start_node, 0);
    if (!p) {
        search_dfs_all(start_node, 0, 0, false);
    }
    else {
        for (int i = 3; i >= 0; i--) {
            if (p[i] < MAX_THREAD) {    //选取合适的产生子线程的层数
                int tmp = p[i];
                search_dfs_all(start_node, 0, i, true);
                break;
            }
        }
    }
    char output[100];
    for (int i = 0; i < new_thread; i++) {  //阻塞直到每个线程都算完
        next[i]->join();
    }
    for (int i = 0; i < new_thread; i++) {  //回收new出的内存
        delete next[i];
    }
    snprintf(output, 100, "%s%d%s", "共有", res_num, "个解");
    create_widget("提示", output);
    File.close();
}
```

dfs主体：

```
void Sudoku::search_dfs_all(SudokuNode node_now, int depth, int recur_depth,
bool if_parent) {
    SudokuNode node_new;
    ...
    for (int j = 1; j <= 9; j++) {  //填入可能最少的格子进入下一层
```

```
        if (!node_now.mark[dis_uni][j]) {
            node_new = node_now;
            if (!node_new.fill(dis_uni, j)) {
                continue;
            }
            if (if_parent && depth == recur_depth) {
                next[new_thread++] = new std::thread(&Sudoku::search_dfs_all,
this, node_new, depth + 1, 0, false);
            }
            else {
                search_dfs_all(node_new, depth + 1, recur_depth, if_parent);
            }
        }
    }
    return;
}
```

输出函数：

```
void Sudoku::output_to_file(SudokuNode& node) {
    int n;
    mut_res_num.lock();
    n = ++res_num;
    mut_res_num.unlock();
    if (n > MAX_COUNT) {    //判断是否需要输出
        return;
    }

    char out[164] = { 0 };
    ......  //对字符串处理使得对应数独的解
    mut_out.lock();
    file_out->operator<<(out);
    mut_out.unlock();
}
```

实验过程

该部分介绍实验中遇到的bug与报错

- 图形界面中文乱码。

  原因：未正确设置文件编码。

  解决方法：更改文件编码为utf-8。

- 改进的dfs不能如期输出结果。

  原因：原来写的回溯逻辑混乱，产生错误。

  解决方法：不考虑回溯，而把数独的结点作为一个类，并在递归的过程中传参。提高了代码的封装性。

- 生成有解数独耗时太长。

  原因：初始给定数字过大时，随机填数字很难顺利生成有解数独。

  解决方法：先随机给定17个数字得到一个有解的数独，再进行挖空。

- ……

实验总结

该部分介绍实验中的收获。

- 初步掌握了利用qt的图形化实现
- 初步掌握了c++面向对象的思想
- 初步掌握了几个通用的搜索算法
- 初步掌握了c++下多线程的实现
- 提高了自行调试代码的能力
- 提高了代码重构的意识