

结对编程

1913632-曾兀 1910842-苗泽昀

编程实现

本次结对编程作业任务要求：

1. 采用C++语言实现
2. 可以使用.Net Framework
3. 运行环境为64-bit Windows 10
4. 代码经过质量分析并消除警告
5. 写出至少10个测试用例来覆盖主要功能，统计覆盖率
6. 使用GitHub来管理源代码和测试用例，根据正常进度及时提交commit

最终完成项目已经上传至[Github](#)。

命令行参数处理

因为本次实验要求编写的数独游戏生成及求解器为控制台程序，因此当用户在使用该程序时所有的参数都是通过命令行输入的，所以实现的程序必须能够对用户输入的参数进行分析，并根据设定的规则来匹配来执行不同的函数完成数独终局、游戏生成或数独游戏求解的操作，对不符合要求的参数应当打印相应的报错信息提示给用户。

这里选择使用Github上的一个开源项目[win-c](#)来实现，该项目实现了对Windows平台下控制台程序参数解析，通过多次调用 `getopt()` 函数即可依次得到用户输入的命令行参数，如果用户输入的参数是合法的，那么就调用已经实现的数独终盘生成、数独游戏生成、数独游戏求解等接口完成对应功能否则则通过命令行输出报错并退出程序。

数独终盘生成

数独形式（左图为数独游戏，右图为数独终盘）：

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

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

求解数独时需要根据9×9盘面上的已知数字，推理出所有剩余空格的数字，并满足每一行、每一列、每一个粗线宫（3×3）内的数字均含1-9且不重复。因此可以得出数独终盘的特点为每一行、每一列、每一个粗线宫（3×3）内的数字均含1-9且不重复。

但是如果从0开始生成一个数独终盘的话，需要首先生成一行（一列）或是一个粗线宫内的数字并且在后续的生成过程中避免生成违背数独规则的终盘，因此这里选择一种比较简单的数度终盘生成算法，从一个九宫格出发，通过矩阵变换得到其他九宫格，从而获得整体数独。

算法具体思路：

首先，在中间的宫格生成一个随机的排列：

			a	b	c			
			d	e	f			
			g	h	i			

然后将中间的宫格向两侧作行变换扩展（需要注意的是原本0-1-2的行排列做变换后只有1-2-0和2-0-1的排列是符合数独规则的，列变换也是一样）：

g	h	i	a	b	c	d	e	f
a	b	c	d	e	f	g	h	i
d	e	f	g	h	i	a	b	c

同时将中间的格子列变换生成上下的宫格：

			c	a	b			
			f	d	e			
			i	g	h			
g	h	i	a	b	c	d	e	f
a	b	c	d	e	f	g	h	i
d	e	f	g	h	i	a	b	c
			b	c	a			
			e	f	d			
			h	i	g			

最后使用左右两个宫格进行行变换，分别上下拓展（也可以使用上下两个宫格进行列变换）：

i	g	h	c	a	b	f	d	e
c	a	b	f	d	e	i	g	h
f	d	e	i	g	h	c	a	b
g	h	i	a	b	c	d	e	f
a	b	c	d	e	f	g	h	i
d	e	f	g	h	i	a	b	c
h	i	g	b	c	a	e	f	d
b	c	a	e	f	d	h	i	g
e	f	d	h	i	g	b	c	a

这样就可以高效地生成合法的数独但这个算法也存在缺陷，生成的数独的宫格之间的相似性较强，符合一定的模式，不能穷举所有的数独，单就本次实验来说是足够的。

```
//生成数独终局
void Sudoku::gen_endgame(Board* board) {
    char row[9];
    random_row_permutation(row); //随机生成一串1-9的数字序列
    //填充第一个宫格
    for (int i = 0; i < 3; i++) {
        (*board)[3][i + 3] = row[i] + '1';
        (*board)[4][i + 3] = row[i + 3] + '1';
        (*board)[5][i + 3] = row[i + 6] + '1';
    }
    //依次填充剩余的宫格
    row_col_extend(board, 3, 3, true);
    row_col_extend(board, 3, 3, false);
    row_col_extend(board, 3, 0, false);
    row_col_extend(board, 3, 6, false);
}
```

数独游戏生成

数独游戏的生成就是在已经生成好的数独终局基础上进行挖“空”处理，需要注意的就是用户输入的参数中的m（游戏难度）、r（挖空范围）和u（是否有唯一解）会对数独游戏的生成产生影响，其中m和r影响的都是最终所生成的数独游戏中的“空”的数量，而u会影响最终所生成的数独游戏是否具有唯一解。

如果不限限制所生成的数独游戏是否具有唯一解，那么数独游戏的生成思路就非常简单，只需要挖出所需数量且不重复的“空”即可，而如果要保证所生成的数独游戏必须具有唯一解，目前没有想到可以在挖“空”的过程中确保生成的数独游戏解唯一，因此在实际实现时采取了一种比较粗暴的解决方式，在挖“空”的过程中，每挖一次，就调用数独游戏求解的接口（下文中讲解具体实现）进行求解判断是否有唯一解，如果有唯一解就继续挖“空”直到达到要求的挖“空”数量；否则就回溯并重新挖“空”。

```
//生成数独游戏
int Sudoku::gen_games(int num, int level, int min_hole_num, int max_hole_num,
bool is_unique) {
    set_hole_num_range(level, &min_hole_num, &max_hole_num);
    while (num > 0) {
        Board board(9, std::vector<char>(9, '$'));
        //生成数独终局
        gen_endgame(&board);

        //随机挖“空”
        unsigned int r;
        rand_s(&r);
        int hole = min_hole_num + r % (max_hole_num - min_hole_num + 1);
        if (dig_hole(&board, hole, is_unique) == false) {
            continue;
        }

        games.boards.push_back(board);
        num--;
    }
    games.output();
    return 0;
}

//挖“空”
bool Sudoku::dig_hole(Board* board, int hole_num, bool is_unique) {
    int count = 0; //记录错误次数，过高(意味着生成了无法求解的数独)则重新生成基础图
    while (hole_num > 0) {
        unsigned int r;
        rand_s(&r);
        int x = r % 9;
        rand_s(&r);
        int y = r % 9;
        if ((*board)[x][y] == '$')
            continue;
        char temp = (*board)[x][y];
        (*board)[x][y] = '$';
        //如果有唯一解要求需要在挖“空”过程中通过求解进行检查
        if (is_unique) {
            if (solve_game(board, is_unique) == true && result.boards.size() ==
1) {
                hole_num--;
                count = 0;
            } else {
                (*board)[x][y] = temp;
            }
        }
    }
}
```

```

        if (count++ > 20) {
            return false;
        }
    }
} else {
    hole_num--;
}
}
return true;
}

```

数独游戏求解

实验中使用深度优先+回溯的思路来求解数独。遵循数独的规则，通过尝试的进行填充，如果发现重复了，那么擦除重新进行新一轮的尝试，直到把整个数组填充完成。

算法步骤：

- 数独首先行，列，还有3×3的宫格内数字是1~9不能重复。
- 声明布尔数组，表明行列中某个数字是否被使用了，被用过视为 true，没用过为 false（该部分使用bitset来进行优化，例如：000001010表示2和4已经存在了）。
- 初始化布尔数组，表明哪些数字已经被使用过了。
- 尝试去填充数组，只要行，列，3×3的宫格内出现已经被使用过的数字，我们就不填充，否则尝试填充。
- 如果填充失败，那么我们需要回溯。将原来尝试填充的地方修改回来。
- 递归直到数独被填充完成。

并且在具体实现时，考虑到在生成数独游戏时可能如果不限制具有唯一解就有可能生成具有多个解的数独游戏，因此在对数独游戏进行求解时会求出所有解。

```

//数独游戏求解、结果打印及保存
int Sudoku::solve_games_and_save_results(std::string path) {
    games.load(path);
    games.output();
    int i = 1;
    for (Board board : games.boards) {
        solve_game(&board, false);

        std::string file_name = "board" + std::to_string(i) + ".s results.txt";
        result.save(file_name);

        std::cout << "----result of board[" << i << "]----\n";
        result.output();
        i++;
    }
    return 0;
}

//求解单个数独游戏
bool Sudoku::solve_game(Board* board, bool is_unique) {
    init_state(board);
    return solve_by_dfs(board, 0, is_unique);
}

//初始化当前数独宫格状态（哪些数字已经被使用过了，哪些位置是“空”）
void Sudoku::init_state(Board* board) {
    memset(&state, 0, sizeof(state));
}

```

```

blanks.clear();
result.boards.clear();

for (int i = 0; i < 9; i++) {
    for (int j = 0; j < 9; j++) {
        if ((*board)[i][j] == '$') {
            blanks.push_back(std::pair<int, int>(i, j));
        } else {
            int n = (*board)[i][j] - '1';
            state.flip(i, j, n);
        }
    }
}

}

//通过dfs+回溯的思路求解数独游戏
bool Sudoku::solve_by_dfs(Board* board, int i, bool is_unique) {
    if (i == blanks.size()) {
        result.boards.push_back(*board);
        if (is_unique && result.boards.size() > 1) {
            return false;
        } else {
            return true;
        }
    }

    int x = blanks[i].first, y = blanks[i].second;

    int mask = state.row[x] | state.col[y] | state.block[(x / 3) * 3 + y / 3];

    for (int num = 0; num < 9; num++) {
        if ((mask & (1 << num)) == 0) {
            state.flip(x, y, num);
            (*board)[x][y] = num + '1';
            if (solve_by_dfs(board, i + 1, is_unique) == false) {
                state.flip(x, y, num);
                (*board)[x][y] = '$';
                return false;
            }
            state.flip(x, y, num);
        }
    }
    (*board)[x][y] = '$';
    return true;
}

```

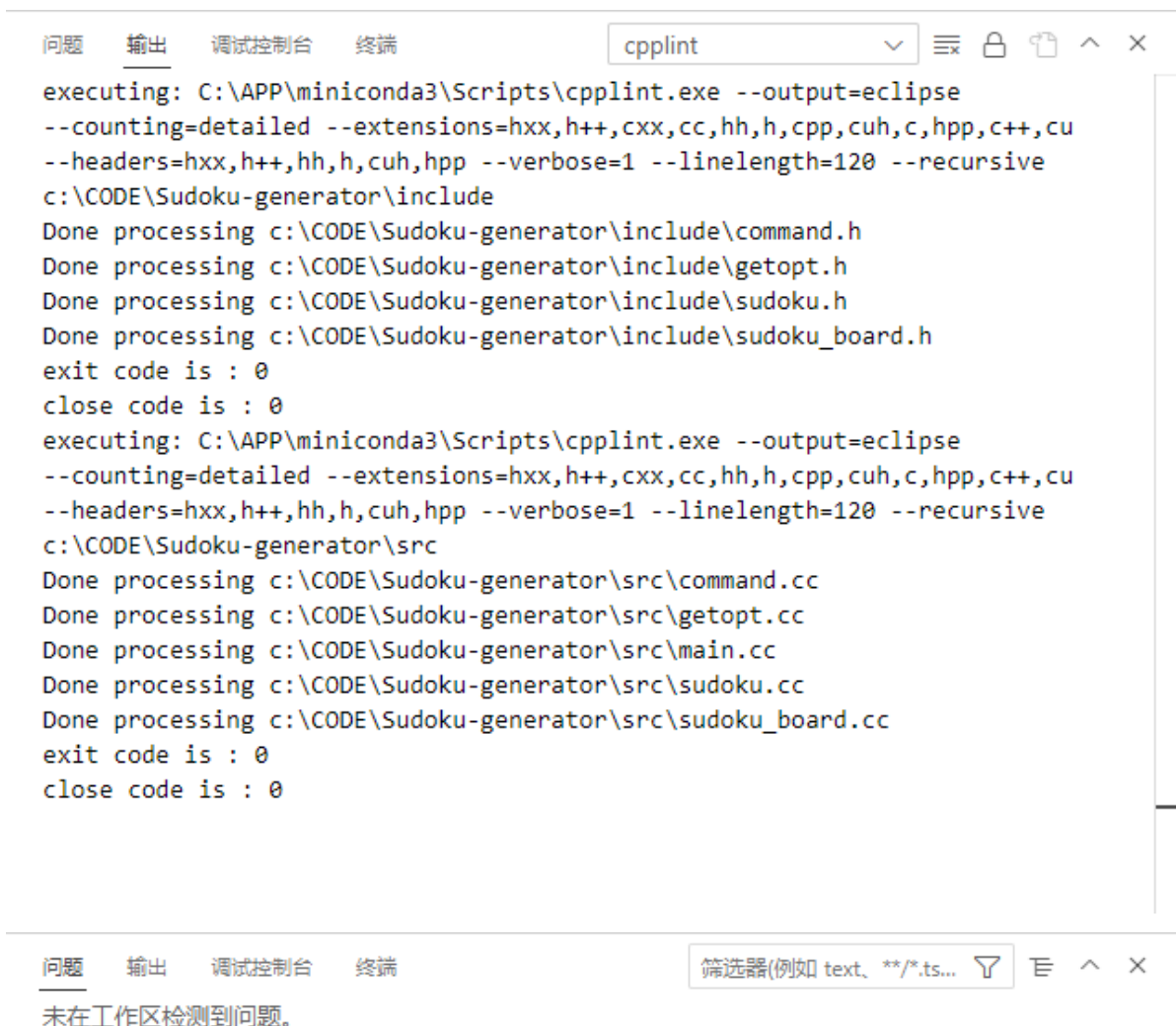
质量分析

静态分析

编程规范检查

编程规范检查使用Cpplint工具（一款开源免费的代码静态检测工具，遵从Google C++代码规范）。

检查结果：



可以看出代码已经完全符合Google C++代码规范。

代码缺陷检查

代码缺陷检查使用Cppcheck工具（一款C/C++代码缺陷静态检查工具）。执行的检查包括：自动变量检查；数组的边界检查；class类检查；过期的函数，废弃函数调用检查；异常内存使用，释放检查；内存泄漏检查，主要是通过内存引用指针；操作系统资源释放检查，中断，文件描述符等；异常STL函数使用检查；代码格式错误，以及性能因素检查等。

检查结果：



可以看出代码中不存在静态检查工具可发现的缺陷。

动态分析

性能测试使用gprof (GNU profiler, 是GNU binutils工具集中的一个工具, 可以分析程序的性能, 能给出函数调用时间、调用次数和调用关系, 找出程序的瓶颈所在。在编译和链接选项中都加入-pg之后, gcc会在每个函数中插入代码片段, 用于记录函数间的调用关系和调用次数, 并采集函数的调用时间) 这里设定调用 sudoku.exe 时的参数设置为 -c 1000 -n 1000 -m 3 -u, 最终得到的性能分析结果为 (只截取了部分关键结果) :

Flat profile:

Each sample counts as 0.01 seconds.

% time	cumulative seconds	self seconds	calls	self s/call	total s/call	name
31.23	1.19	1.19				_mcount_private


```

26.51      2.20      1.01      60816      0.00      0.00
Sudoku::solve_by_dfs(std::vector<std::vector<char, std::allocator<char> >,
std::allocator<std::vector<char, std::allocator<char> > >*>, int, bool)
21.78      3.03      0.83      __fentry__
12.34      3.50      0.47 115782056      0.00      0.00  Sudoku::State::flip(int,
int, int)
2.62      3.60      0.10 121567285      0.00      0.00
std::vector<std::vector<char, std::allocator<char> >,
std::allocator<std::vector<char, std::allocator<char> > >*>::operator[](unsigned
long long)
2.36      3.69      0.09 121567285      0.00      0.00  std::vector<char,
std::allocator<char> >::operator[](unsigned long long)
1.57      3.75      0.06 112574212      0.00      0.00  std::vector<std::pair<int,
int>, std::allocator<std::pair<int, int> > >::operator[](unsigned long long)
0.52      3.77      0.02      60816      0.00      0.00
Sudoku::init_state(std::vector<std::vector<char, std::allocator<char> >,
std::allocator<std::vector<char, std::allocator<char> > >*>)
0.26      3.78      0.01 5224612      0.00      0.00  std::pair<int, int>&&
std::forward<std::pair<int, int> >(std::remove_reference<std::pair<int, int>
>::type&)
0.26      3.79      0.01      690423      0.00      0.00
__gnu_cxx::new_allocator<char>::allocate(unsigned long long, void const*)
0.26      3.80      0.01      688383      0.00      0.00  std::vector<char,
std::allocator<char> >::begin() const
0.26      3.81      0.01      60816      0.00      0.00
Sudoku::solve_game(std::vector<std::vector<char, std::allocator<char> >,
std::allocator<std::vector<char, std::allocator<char> > >*>, bool)
.....

```

可以看出数独终局的生成在经过优化后的时间开销已经很低，而关键的性能瓶颈在 `solve_by_dfs()` 函数，因为在生成具有唯一解的数独游戏和数独游戏的解答中都会使用到该函数，该函数还是递归实现。因此在后续的工作中应当着重优化 `solve_by_dfs()` 函数的时间复杂度，寻找可以生成有唯一解的数独游戏和求解数独游戏的更优算法。

单元测试

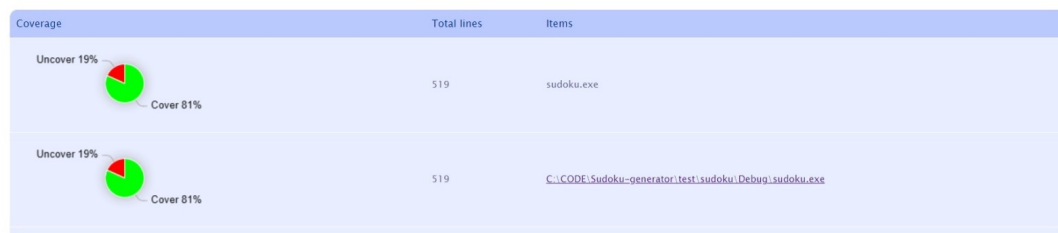
单元测试使用OpenCppCoverage（Windows平台下开源的C++代码覆盖率工具），在实验中共设置了38个测试样例（包括参数和文件内容合法的测试样例和参数和文件内容合法的测试样例，全部样例可查看[test.bat](#)文件），这里列出部分测试样例：

编号	参数	预期反馈	实际反馈	是否通过
1	-c 50	生成50个数独终局	生成50个数独终局	通过
2	-n 50 -m 3	生成50个难度为3的数独游戏	生成50个难度为3的数独游戏	通过
3	-n 50 -m 3 -r 25~50	生成50个难度为3的数独游戏	生成50个难度为3的数独游戏	通过
4	-n 50 -m 3 -u	生成50个难度为3且解唯一的数独游戏	生成50个难度为3且解唯一的数独游戏	通过
5	-s .\board_1.txt	给出.\board_1.txt中的数独游戏的解答	给出.\board_1.txt中的数独游戏的解答	通过
6	-t a	报错并退出	报错并退出	通过
7	-c 1000001	报错并退出	报错并退出	通过
8	-n 50 -m 4	报错并退出	报错并退出	通过
9	-n 50 -r 25	报错并退出	报错并退出	通过
10	-s .\bad_board_1.txt	报错并退出	报错并退出	通过

最终的代码覆盖率为81%：








sudoku.exe

Warning: Your program has exited with error code: -1



查看具体每个代码文件的覆盖率：

sudoku.exe

Coverage	Total lines	Items
 Uncover 19% Cover 81%	519	C:\CODE\Sudoku-generator\test\sudoku\Debug\sudoku.exe
 Uncover 63% Cover 37%	140	C:\CODE\Sudoku-generator\test\sudoku\sudoku\src\getopt.cc
 Uncover 3% Cover 97%	176	C:\CODE\Sudoku-generator\test\sudoku\sudoku\src\sudoku.cc
 Uncover 3% Cover 97%	74	C:\CODE\Sudoku-generator\test\sudoku\sudoku\src\sudoku_board.cc
 Uncover 2% Cover 98%	116	C:\CODE\Sudoku-generator\test\sudoku\sudoku\src\command.cc
 Uncover 0% Cover 100%	5	C:\CODE\Sudoku-generator\test\sudoku\sudoku\include\sudoku.h
 Uncover 0% Cover 100%	8	C:\CODE\Sudoku-generator\test\sudoku\sudoku\src\main.cc

可以看出未覆盖的代码中绝大部分都是命令行参数处理中使用的开源项目[win-c](#)的代码文件，而后续编写的代码文件中未覆盖的代码只有“文件打开失败”报错、部分if-else分支中的else（不是else分支的具体代码，单就else这一行代码，可能是软件存在部分bug）。