2N皇后 程序说明

本实验中所有程序都用C编写（非C++），利用C编译器可以在一定程度上提高程序运行的效率。

CSP

CSP的大致过程和书中一致，即每次选择一行，将这一行的棋子移动到冲突最少的位置。这里，有以下几点值得说明：

1、在保存棋盘的时候，只需要使用大小为N的数组（N为皇后数量）即可，因为可用数组的下标代表皇后的行数，数组的内容代表皇后所在的列数，而不需要大小为N*N的数组（因为每一行只能有一个皇后）。

2、每一次计算某一个格子的冲突数时，只需要`O(1)`的时间就可以完成。只需要利用额外的三个数组`lineQNum[N]`，`posDiaQNum[2 * N - 1]`，`negDiaQNum[2 * N - 1]`分别记录某一列的总皇后数量，主对角线的皇后数量和次对角线的皇后数量，即可在`O(1)`的时间得到某一方格的冲突数：

```
int GetConflict(int row, int line)
{
    int a = lineQNum[line];
    int b = posDiaQNum[boardScale - line + row - 1];
    int c = negDiaQNum[line + row];
    return a + b + c;
}
```

其中，`boardScale`的值等于N。除此之外，类似地，维护这三个数组所需的时间也是`O(1)`。这样，每次在进行某一行中皇后的移动时则可以在`O(n)`的时间内完成。

3、判断是否到达最终状态可以在`O(1)`时间内完成。只需要维护一个变量`conflictSum`，每次迭代完成之后，只需要使`conflictSum -= 2 * (preMin - min)`，即可得到新的状态下的总冲突数量。当

`conflictSum`等于0时，算法即终止。

4、除此之外，为了防止算法在某些特殊的情况下陷入循环，我给算法加上了一些随机因素，例如每次在出现相同的最小代价时近似随机地选择一个值，程序开始时随机初始化棋盘等。这些随机因素可能会影响算法的运行效率。

另外，还有一点特别需要说明：

本次实验中，我在计算2N皇后的时候，只计算了N皇后下的可行解，然后再想办法将这个解扩展成2N皇后的情况。具体如下：

1、如果N是偶数，那么只需要将N皇后的解做轴对称即可得到另一个合理的解，将这两个解拼接起来即可得到2N皇后的最终解。这是可以证明的：对称之后原来N皇后的解在对称位置上不可能有棋子，因为一行只能有一个皇后。

2、N为奇数时，上面的方法将不起作用，因为中心线（即对称轴）上必定会有棋子，这颗棋子用上述的方法处理会导致重叠。因此，我采取的办法是在算法进行的过程中，不允许算法在最长的主对角线上放置棋子。之后，将得到的解沿主对角线做对称，拼接之后得到新的解即可。事实证明，增加这样的约束时候基本不会影响到算法的运行时间，反而可能使算法的运行时间变快，因为在有选择的时候，程序不应该往最长的主对角线上放置棋子：最长主对角线可能引起的冲突数是最多的。

我在模拟退火算法中也采取了这种方法，这样由N皇后的解得到2N皇后的解只需要`O(N)`的时间。当然，这样做似乎有些偏离题意，因此我提供了一个最初版本的源代码，这个版本中实现了真正对2N皇后问题进行求解的算法，同时计算冲突数采用的算法也和上文中不同，但是完成一行皇后移动的时间复杂度仍然是`O(n)`（当然常数项会偏大，因此之后我更换了实现方法）。这个程序`2NQueen_CSP_pre`的执行效率会比`2NQueen_CSP`低得多，仅供参考。

模拟退火

这个程序中同样采用了上文中前三点的优化方式。除此之外，我实现模拟退火算法的时候选择的行数依次递增，只有列数是随机的（即受模拟退火算法的影响，采用随机选择的方式）。模拟退火算法的实现和书中的伪代码一致，这里不赘述。在实现时，我将迭代次数的倒数作为温度T，冲突数的变化量作为评估标准ΔE。采用了上文的优化方式之后，程序的总空间复杂度为`O(N)`，每一次迭代的时间复杂度为`O(1)`。

运行说明

我的编译环境是windows10 with visual studio 2015 community，所有程序直接运行即可。由于有随机因素的影响，因此每次运行可能得到不同的结果，运行时间可能也会有很大差别。在我的测试环境下（CPU：i5-4200H），模拟退火算法和CSP算法大致能够解决的问题规模都在10000个皇后左右的量级（5分钟之内）。