Задача категории Б

Благодарный Артём

28 января 2025 г.

Введение

Я начну с того, что "Задача категории Б"является идентичной задачей расстановки ферзей или, по-другому, "**N-Queens problem**". **Цель проста: на шахматной доске NxN расположить ферзей так, чтобы ни один ферзь не мог атаковать другого**. Входные данные: N $\leq$ 2000, выходные данные: расстановка ферзей(поле NxN, где на свободных позициях - 0, а на позициях ферзя - 1). Эта задача имеет много решений, в зависимости от разных N, но в данной статье я рассмотрю мои попытки решения этой задачи и наиболее популярные из них:

- **Backtracking** - возврат к предыдущей расстановки
- **MinConflict** - метод минимальных конфликтов
- **Explicit solution** -точное решение для любого N

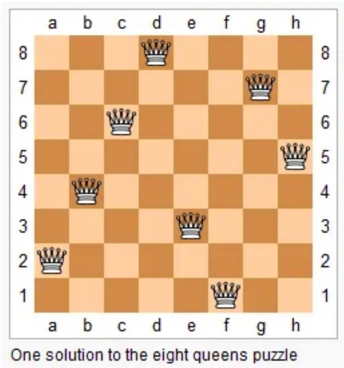

Рис. 1: Пример расстановки ферзей для поля 8x8

Первые попытки

После прочтения статьи на habr про эту задачу и, увидев, что её можно решить однозначно, я стал пытаться высчитать индексы, но мои расчёты были верны только в 4 из 6 случаев, поэтому, сдавшись, я решил вернуться к методу умного перебора - **Backtracking**.

Прочитав английскую статью и посмотрев пару видео на YouTube: вот и вот, я решил попробовать самостоятельно написать код.

```
class ProblemCategoryB:
    def __init__(self, m):
        self.n = m

    def solution(self):
        matrix = [[0] * self.n for _ in range(self.n)]
        columns, diagonal1, diagonal2 = set(), set(), set()
        flag = True
```

```
        def backtracking(row):
            nonlocal flag
            if flag:
                if row == self.n:
                    for res in matrix:
                        print("".join(map(str, res)))
                    flag = False
                    return
                for j in range(self.n):
                    if ((j in columns) or
                       (row + j) in diagonal1 or
                        (row - j) in diagonal2):
                        continue
                    columns.add(j)
                    diagonal1.add(row + j)
                    diagonal2.add(row - j)
                    matrix[row][j] = 1
                    backtracking(row + 1)
                    columns.remove(j)
                    diagonal1.remove(row + j)
                    diagonal2.remove(row - j)
                    matrix[row][j] = 0

        backtracking(0)


task = ProblemCategoryB(int(input()))
task.solution()
```

Основная идея: у нас есть две диагонали(одну считаем как i + j, а вторую как i - j), столбцы и строки. Мы будем итерироваться по строкам и каждый раз проверять: можно ли на позицию (i, j) поставить ферзя, если можно,то обновляем columns, diagonal1 и diagonal2 и переходим к новой строке. Когда просмотрели все возможные позиции, удаляем данные о местоположении ферзя, тем самым делая возврат, и итерируемся дальше.

В моём случаем у меня программа быстро($<$ 1 сек) находила ответ для $N \leq 29$. Эту программу можно улучшить, но почитав в интернете, я нашёл, что она будет работать хорошо при $N \leq 500$, а у нас по условию $N \leq 2000$. Идеи для улучшения: использовать эвристику и отсечение по рекорду. Эвристика: выбирать позицию ферзя такую, которая оставляет больше свободных клеточек. Отсечение по рекорду: посчитать сколько можно ещё поставить ладей либо слонов, после того как поставили ферзя. Если сумма ферзей и ладей/слонов будет $<$ N, то пропускаем данную позицию.

Это просто гениально!

Поискав новую информацию в интернете, мне попался слиток золота - вот эта статья! В данной статье рассказывается метод минимальных конфликтов, который может быстро получить ответ для $N \leq 1000000$! Сначала определим, что такое конфликт. Конфликт - это ситуация на доске, когда два ферзя атакуют друг друга. Основная идея **MinConflict** очень проста:

1. Просматриваем тех ферзей, которые имеют конфликты.
2. Случайным образом выбираем среди таких ферзей одного.
3. Пытаемся переставить ферзя на строку, на которой конфликт будет наименьшим.
4. Обновляем доску.
5. Повторяем, пока количество есть конфликты.

Но, чтобы код работал быстро, нужно определять строку с минимальным количеством конфликтов за O(N). Чтобы выполнить обновление за время O(N), мы перебираем всех остальных ферзей и удаляем конфликты, которые были вызваны старой позиции ферзя в строке, и добавляем конфликты, вызванные после его перемещения в новую строку. Основная функция:

```
def min_conflict(self, N, max_steps=10 ** 6):
    queens = self.initialize_queens(N)
    conflicts = self.build_conflicts(queens)
    for i in range(max_steps):
        if sum(conflicts) == 0:
            return queens, i
        col = self.pick_position(
            conflicts, N, lambda x: x > 0)
        row_confs = self.row_conflicts(queens, N, col)
        min_conf = min(row_confs)
        new_row = self.pick_position(
            row_confs, N, lambda x: x == min_conf)
        self.update_conflicts(
            queens, conflicts, col, new_row)
        queens[col] = new_row
    return queens, max_steps
```

На практике метод **MinConflict** широко используется. Одним из примеров является то, что НАСА использовало его, чтобы сократить время, необходимое им для планирования всех запросов на использование своих телескопов, чтобы не было конфликтов во времени. Использование **MinConflict** сократило общее время планирования с почти дня до примерно 10 минут. Это очень круто!

Точное решение

После нахождения такого прекрасного решения, я был в восторге, но меня не покидала мысль, что есть точное решение этой задачи. Покопавшись в интернете, мне всё-таки удалось найти те самые индексы, которые давали бы решение задачи за O(1). Первая статья, которая давала точный ответ была вот эта. В ней было сказано, что впервые точное решение было опубликовано аж в 1969 году! Покопавшись ещё, я нашёл прекрасную статью от NASA, в которой был прекрасно расписан алгоритм с доказательством.

Вот код, основанный на вычислении индекса:

```
def chessMethod1(self, n):
    return lambda i: 2 * i if i <= n / 2 else 2 * (i - n // 2) - 1


def chessMethod2(self, n):
    return lambda i: 1 + (2 * i + n // 2 - 3) % n if\
        i <= n / 2 else n - (2 * (n + 1 - i) + n // 2 - 3) % n


def finalChess(self):
    n = self.n
    if n % 2 == 0 and (n % 6 != 2):
        return self.chessMethod1(n)
    elif n % 2 == 0:
        return self.chessMethod2(n)
    elif (n - 1) % 6 != 2:
        return lambda j: n if j == n else self.chessMethod1(n - 1)(j)
    else:
        return lambda j: n if j == n else self.chessMethod2(n - 1)(j)
```

Итог

Задача **N-Queens** — популярное алгоритмическое упражнение в информатике. Мы рассмотрели разные подходы к решению данной задачи. Какой подход выбрать, это дело читателя. Вот ссылка на репозиторий Git, в котором можно посмотреть разные подходы к решению данной задачи.