

Algorytmy z nawrotami

Sprawozdanie z laboratorium 10 – Piotr Sarna LK1

Cel ćwiczenia

Podczas zajęć zapoznaliśmy się ze sposobem rozwiązywania problemów wykorzystując algorytmy z nawrotami. Następnie wykorzystaliśmy go do rozwiązania problemu „n-hetmanów”.

Wstęp teoretyczny

Algorytmy z nawrotami służą do generowania wszystkich rozwiązań danego problemu, poprzez próbowanie wszystkich możliwości, oraz rezygnację i cofanie się, gdy stwierdzi, że dana możliwość na pewno nie prowadzi do rozwiązania.

Metoda ta potrafi być znacząco szybsza od wyczerpującego wyszukiwania rozwiązań, ponieważ odcinając jedno z nich potencjalnie odcina także wiele innych.

Algorytmy z nawrotami bazują na rekurencji, można je wykorzystać do rozwiązania np. następujących problemów:

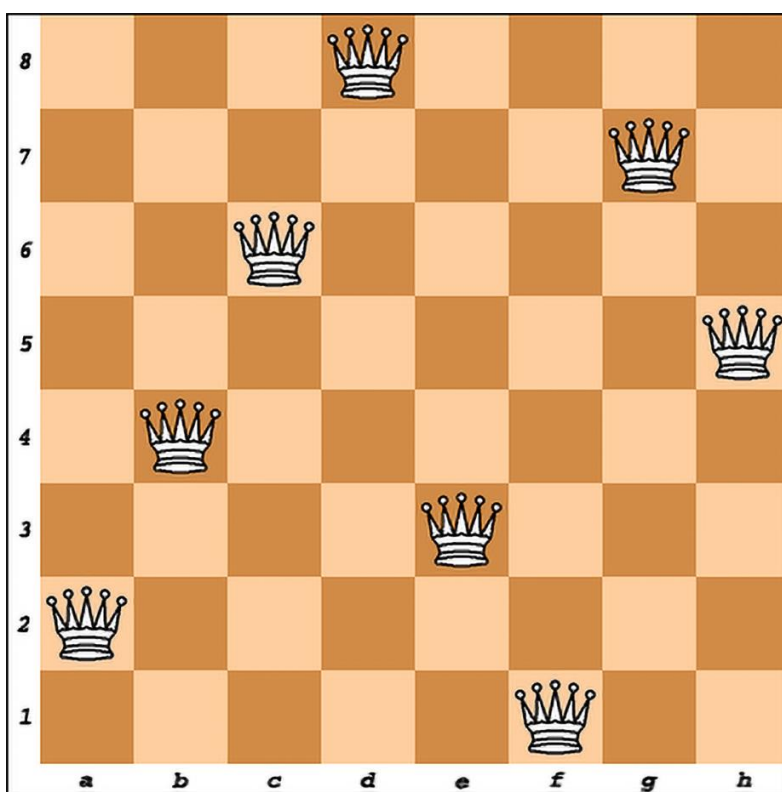
- Problem komiwojażera
- Problem skoczka szachowego
- Problem n-hetmanów

Opis algorytmu

Problem n-hetmanów polega na znalezieniu takiego ustawienia n hetmanów na szachownicy o wymiarach $n \times n$, aby żaden z nich nie szachował innego.

Oznacza to, że każda z figur nie może stać w wierszu, kolumnie ani przekątnej innej.

Przykład rozwiązania problemu dla 8 hetmanów:



Źródło: <https://mlodytechnik.pl/eksperymenty-i-zadania-szkolne/szachy/31020-problem-osmiu-hetmanow>

Przebieg działania algorytmu z nawracaniem dla problemu n-hetmanów:

- Ustawiamy pierwszego hetmana w pierwszej kolumnie pierwszego wiersza
- Szukamy w wierszu niżej miejsca, które nie jest szachowane przez żadnego z uprzednio postawionych hetmanów, jeśli znajdziemy takie miejsce – wstawiamy w nie nowego hetmana
- Jeśli w danym wierszu nie znajdziemy żadnego miejsca, w którym moglibyśmy wstawić nowego hetmana, cofamy się do poprzedniego wiersza i w nim przesuwamy hetmana na kolejną, nieszachowaną pozycję
- Powtarzamy cykl tak długo, aż w ostatnim wierszu postawimy ostatniego hetmana – możemy wtedy zakończyć działanie i wypisać szachownicę z ułożonymi hetmanami

Implementacja rozwiązania problemu n-hetmanów za pomocą algorytmu z nawrotami w C++.

```
void nQueenProblem(const int& count) {
    bool** board = new bool*[count];
    for (int i = 0; i < count; i++) {
        board[i] = new bool[count] {false};
    }

    solveQueens(board, 0, count);

    for (int i = 0; i < count; i++) {
        delete[] board[i];
    }
    delete[] board;
}
```

Funkcja nQueenProblem przyjmuje jako argument ilość hetmanów, na jej podstawie tworzy macierz n x n, która będzie reprezentowała planszę, na której będą znajdowały się hetmany.

```

bool isSafe(bool** board, const int& row, const int& col, const int& count) {
    for (int i = 0; i < row; i++) {
        if (board[i][col]) {
            return false;
        }

        for (int j = 0; j < count; j++) {
            if (board[i][j] && std::abs(row - i) == std::abs(col - j)) {
                return false;
            }
        }
    }

    return true;
}

```

Funkcja isSafe sprawdza, czy dla aktualnego ułożenia hetmanów na planszy, można na wybranym miejscu postawić nowego hetmana tak, aby nie był on szachowany.

```

void solveQueens(bool** board, int row, const int& count) {
    if (row == count) {
        printBoard(board, count);
        return;
    }

    for (int col = 0; col < count; col++) {
        if (isSafe(board, row, col, count)) {
            board[row][col] = true;
            solveQueens(board, row + 1, count);
            board[row][col] = false;
        }
    }
}

```

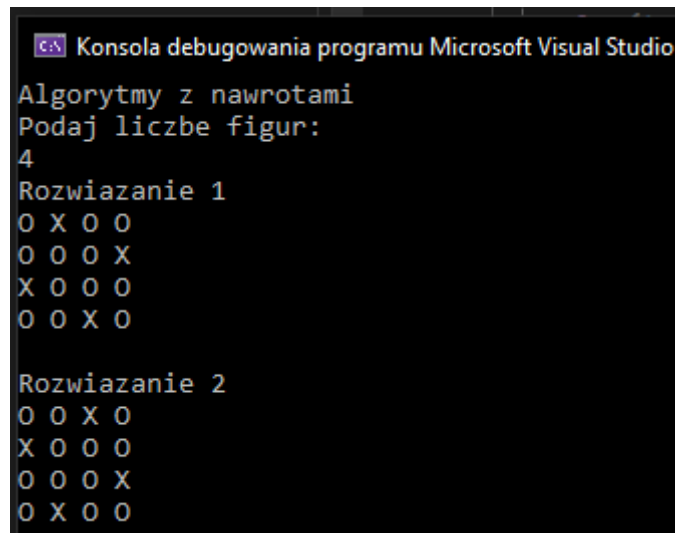
Funkcja solveQueens działa rekurencyjnie. Pracuje na zadanym początkowo wierszu – początkowo jest to wiersz o indeksie „0”.

Początkowy warunek rekurencyjny sprawdza, czy doszliśmy do ostatniego wiersza w macierzy. Jeśli tak, kończymy rekurencję i wypisujemy planszę wywołując funkcję printBoard.

W przeciwnym wypadku, przesuwamy się po kolejnych elementach w zadanym wierszu. Jeśli możemy na nim postawić

nowego hetmana, zapisujemy go w macierzy i wywołujemy rekurencyjnie funkcję solveQueens dla kolejnego wiersza.

Prezentacja działania mojej implementacji w C++ dla $n = 4$:



```
Konsola debugowania programu Microsoft Visual Studio
Algorytmy z nawrotami
Podaj liczbe figur:
4
Rozwiazanie 1
0 X 0 0
0 0 0 X
X 0 0 0
0 0 X 0

Rozwiazanie 2
0 0 X 0
X 0 0 0
0 0 0 X
0 X 0 0
```

Wnioski

Nasz algorytm w najgorszym przypadku ma złożoność obliczeniową $O(n!)$, ponieważ:

- Pierwszy wiersz: n możliwości
- Drugi wiersz: $n - 1$ możliwości

...

- N -ty wiersz: 1 możliwość

Lecz w rzeczywistości dzięki nawrotom, jego złożoność obliczeniowa wypada znacznie lepiej.

Bibliografia

<https://mlodytechnik.pl/eksperymenty-i-zadania-szkolne/szachy/31020-problem-osmiu-hetmanow>

<https://pages.mini.pw.edu.pl/~kaczmarskik/MiNIWyklady/sieci/hetmany.html>