

Sztuczna inteligencja i inżynieria wiedzy laboratorium

## **Sprawozdanie 2**

*Problem spełniania ograniczeń*

Arkadiusz Marcinowski

228160

W8, Informatyka

# 1. Wstęp

Celem zadanie było zapoznanie się z podstawowymi algorytmami stosowanymi do rozwiązywania problemów spełniania ograniczeń (ang. Constraint Satisfaction Problem, CSP), poprzez własnoręczną implementację i zbadanie ich właściwości.

Problem N-hetmanów oraz kwadratu łacińskiego zdefiniowano jako problem CSP i rozwiązano wykorzystując algorytmy sprawdzenia w przód oraz przeszukiwania przyrostowego z powracaniem dla różnych wartości N, gdzie N definiuje wielkość wybranego problemu.

## 2. Podejście do problemu

### 2.1. Przeszukiwanie przyrostowe

- przeszukiwanie w głąb, każdy krok to ustalenie wartości jednej zmiennej
- kolejność przypisywania zmiennych jest ustalona
- jeśli ustalenie kolejnej zmiennej jest niewykonalne bez łamania więzów następuje powrót, tzn. cofnięcie niektórych przypisań

### 2.2. Przeszukiwanie w przód

Nadając wartość pewnej zmiennej usuwamy z dziedziny każdej nieokreślonej jeszcze zmiennej te wartości, które są sprzeczne z wartościami zmiennych już określonych. Jeśli w wyniku usunięcia któraś z dziedzin stanie się pusta, wykonujemy nawrót.

## 3 Test działania

Przeprowadzono test działania programu dla problemu N-hetmanów, z użyciem oprogramowania *YourKit Java Profiler*.

Wyniki pokazały, że najbardziej kosztowną metodą jest `goRecursive()`. Poniższy screen przedstawia wyniki przed i po optymalizacji.

PRZED:

	Call Tree	Time (ms)
<All threads>		7,473 100 %
com.intellij.rt.execution.application.AppMainV2\$1.run()		3,738 50 %
com.si.Main.main(String[])		3,734 50 %
Main.java:14 com.si.HetmanBacktracking.go()		2,593 35 %
HetmanBacktracking.java:90 com.si.HetmanBacktracking.goRecursive(int)		1,437 19 %
HetmanBacktracking.java:105 com.si.HetmanBacktracking.goRecursive(int)		1,437 19 %
HetmanBacktracking.java:105 com.si.HetmanBacktracking.goRecursive(int)		1,437 19 %
HetmanBacktracking.java:105 com.si.HetmanBacktracking.goRecursive(int)		1,437 19 %
HetmanBacktracking.java:105 com.si.HetmanBacktracking.goRecursive(int)		1,437 19 %
HetmanBacktracking.java:105 com.si.HetmanBacktracking.goRecursive(int)		1,437 19 %
HetmanBacktracking.java:78		1,078 14 %
HetmanBacktracking.java:82		46 1 %
HetmanBacktracking.java:80		31 0 %
Main.java:21 com.si.HetmanForwardChecking.go()		1,140 15 %
HetmanForwardChecking.java:78 com.si.HetmanForwardChecking.goRecursive(int, ArrayList)		1,140 15 %
HetmanForwardChecking.java:95 com.si.HetmanForwardChecking.goRecursive(int, ArrayList)		1,140 15 %
HetmanForwardChecking.java:95 com.si.HetmanForwardChecking.goRecursive(int, ArrayList)		1,140 15 %
HetmanForwardChecking.java:95 com.si.HetmanForwardChecking.goRecursive(int, ArrayList)		1,140 15 %
HetmanForwardChecking.java:95 com.si.HetmanForwardChecking.goRecursive(int, ArrayList)		1,140 15 %

	Method	Time (ms)	Own Time (ms)
⇒ com.intellij.rt.execution.application.AppMainV2\$1.run()	AppMainV2.java	3,738 50 %	3,738
→ com.si.Main.main(String[])	Main.java	3,734 50 %	0
→ com.si.HetmanBacktracking.go()	HetmanBacktracking.java	2,593 35 %	1,156
→ com.si.HetmanBacktracking.goRecursive(int)	HetmanBacktracking.java	1,437 19 %	1,437
→ com.si.HetmanForwardChecking.go()	HetmanForwardChecking.java	1,140 15 %	0
→ com.si.HetmanForwardChecking.goRecursive(int, ArrayList)	HetmanForwardChecking.java	1,140 15 %	31
→ com.si.HetmanForwardChecking.getDomain(int)	HetmanForwardChecking.java	1,109 15 %	1,046
→ com.si.HetmanForwardChecking.isSafe(int, int)	HetmanForwardChecking.java	31 0 %	31

PO OPTYMALIZACJI:

	Method	Time (ms)	Own Time (ms)
⇒ com.intellij.rt.execution.application.AppMainV2\$1.run()	AppMainV2.java	1,702 51 %	1,702
→ com.si.Main.main(String[])	Main.java	1,656 49 %	0
→ com.si.HetmanBacktracking.go()	HetmanBacktracking.java	1,046 31 %	0
→ com.si.HetmanBacktracking.goRecursive(int)	HetmanBacktracking.java	1,046 31 %	1,046
→ com.si.HetmanForwardChecking.go()	HetmanForwardChecking.java	609 18 %	0
→ com.si.HetmanForwardChecking.goRecursive(int, ArrayList)	HetmanForwardChecking.java	609 18 %	93
→ com.si.HetmanForwardChecking.getDomain(int)	HetmanForwardChecking.java	515 15 %	515

Call Tree		Time (ms)
<All threads>		3,358 100 %
com.intellij.rt.execution.application.AppMainV2\$1.run()		1,702 51 %
com.si.Main.main(String[])		1,656 49 %
Main.java:14 com.si.HetmanBacktracking.go()		1,046 31 %
HetmanBacktracking.java:65 com.si.HetmanBacktracking.goRecursive(int)		1,046 31 %
HetmanBacktracking.java:80 com.si.HetmanBacktracking.goRecursive(int)		1,046 31 %
HetmanBacktracking.java:80 com.si.HetmanBacktracking.goRecursive(int)		1,046 31 %
HetmanBacktracking.java:80 com.si.HetmanBacktracking.goRecursive(int)		1,046 31 %
HetmanBacktracking.java:80 com.si.HetmanBacktracking.goRecursive(int)		1,046 31 %
HetmanBacktracking.java:80 com.si.HetmanBacktracking.goRecursive(int)		1,015 30 %
HetmanBacktracking.java:80 com.si.HetmanBacktracking.goRecursive(int)		1,015 30 %
HetmanBacktracking.java:80 com.si.HetmanBacktracking.goRecursive(int)		984 29 %
HetmanBacktracking.java:80 com.si.HetmanBacktracking.goRecursive(int)		843 25 %
HetmanBacktracking.java:80 com.si.HetmanBacktracking.goRecursive(int)		609 18 %
HetmanBacktracking.java:80 com.si.HetmanBacktracking.goRecursive(int)		421 13 %
HetmanBacktracking.java:80 com.si.HetmanBacktracking.goRecursive(int)		234 7 %
HetmanBacktracking.java:74		187 6 %
HetmanBacktracking.java:74		187 6 %
HetmanBacktracking.java:74		234 7 %
HetmanBacktracking.java:74		140 4 %
HetmanBacktracking.java:74		31 1 %
HetmanBacktracking.java:74		31 1 %
Main.java:21 com.si.HetmanForwardChecking.go()		609 18 %
HetmanForwardChecking.java:78 com.si.HetmanForwardChecking.goRecursive(int, ArrayList)		609 18 %
HetmanForwardChecking.java:95 com.si.HetmanForwardChecking.goRecursive(int, ArrayList)		609 18 %
HetmanForwardChecking.java:95 com.si.HetmanForwardChecking.goRecursive(int, ArrayList)		609 18 %

## 4. Badania

### 4.1 N-Hetmanów

Zależność czasu (nano s) pracy algorytmu od wielkości problemu, dla N-Hetmanów.

N	1	2	3	4	5	6	7	8	9	10	11	12
BT	8033	3570	5355	15172	57120	153062	413224	2845712	2173666	10025369	32850867	149391208
FC	16511	7139	13834	70953	164665	262393	1100889	2072368	3302223	1631878	32604539	159431304

N	13	14	15	16
BT	916615758	798069985	31401111600	2,26001E+11
FC	798069985	4689280495	31234051270	2,01241E+11

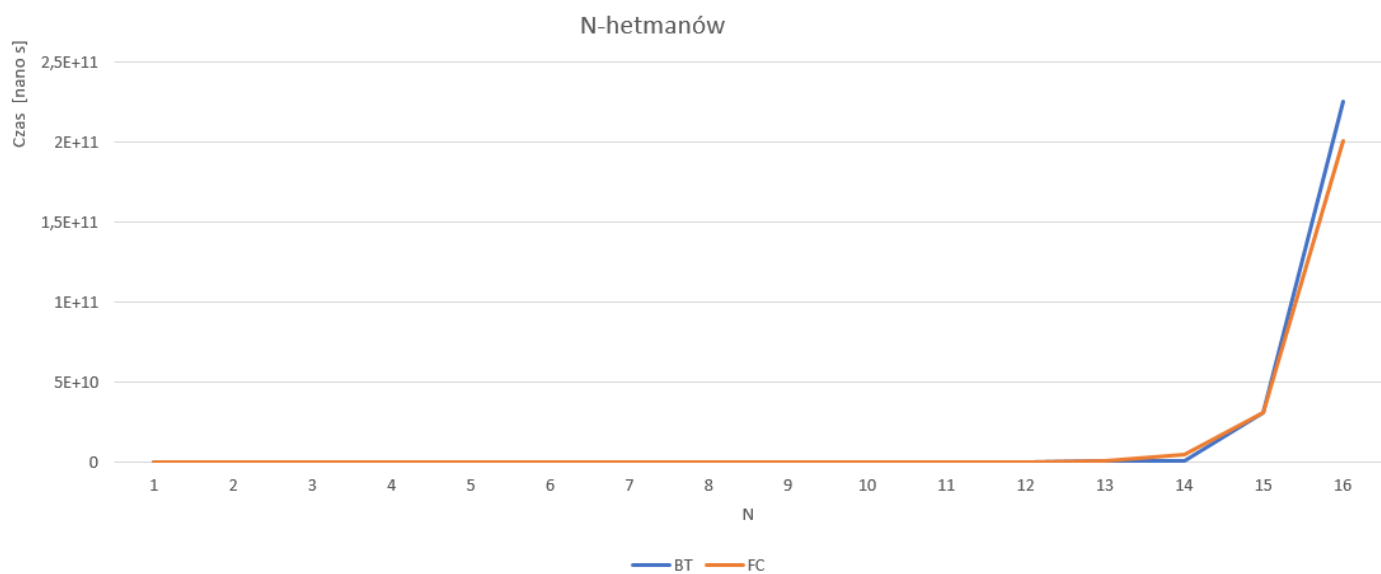
Zauważono, że do N mniejszego od 10, **przeszukiwanie przyrostowe** jest szybszym sposobem na znalezienie wszystkich rozwiązań. Dla N większego od 10, to **sprawdzanie w przód** notuje lepsze czasy.

Zależność liczby odwiedzonych węzłów przez algorytm, od wielkości problemu, dla N-Hetmanów.

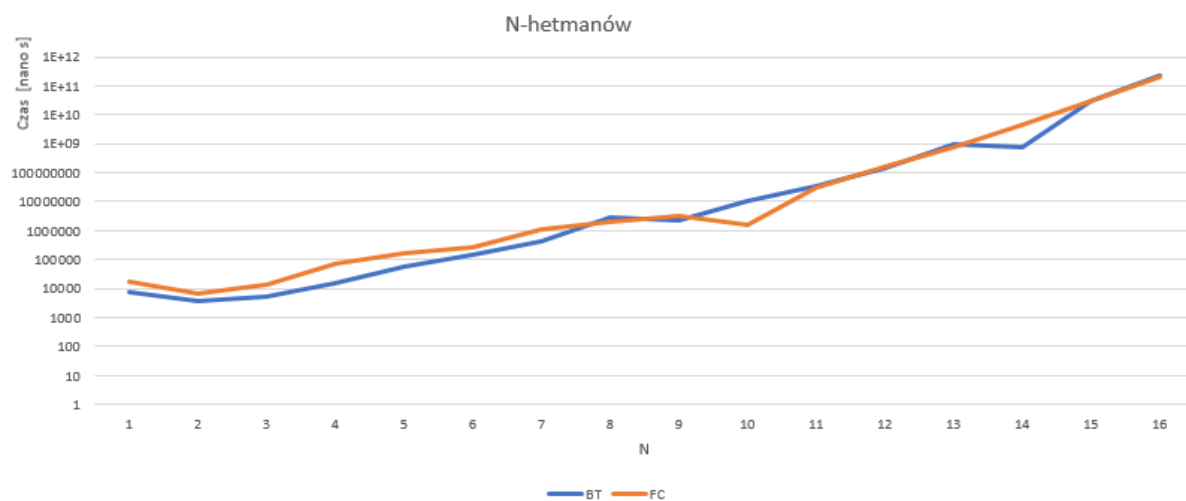
N	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
BT	2	3	6	17	54	153	552	2057	8394	35539	166926	856189	4674890	27358553	171129072	1141190303
FC	2	1	3	13	50	107	398	1413	5810	23489	108530	555659	3031950	17693457	110889484	740447439

Zauważono, że **algorytm sprawdzania w przód** odwiedza dużo więcej węzłów, niż **algorytm przyrostowy**. Różnica jest niemal dwukrotna.

Analizując dwa powyższe zestawienia stwierdzono również, że **algorytm sprawdzania wprzód** spędza więcej czasu w każdym węźle, niż ma to miejsce w przypadku **algorytmu przyrostowego**.



W skali logarytmicznej:



## 4.2 Kwadrat łaciński

Zależność czasu (ms) pracy algorytmu od wielkości problemu, dla Kwadratu łacińskiego.

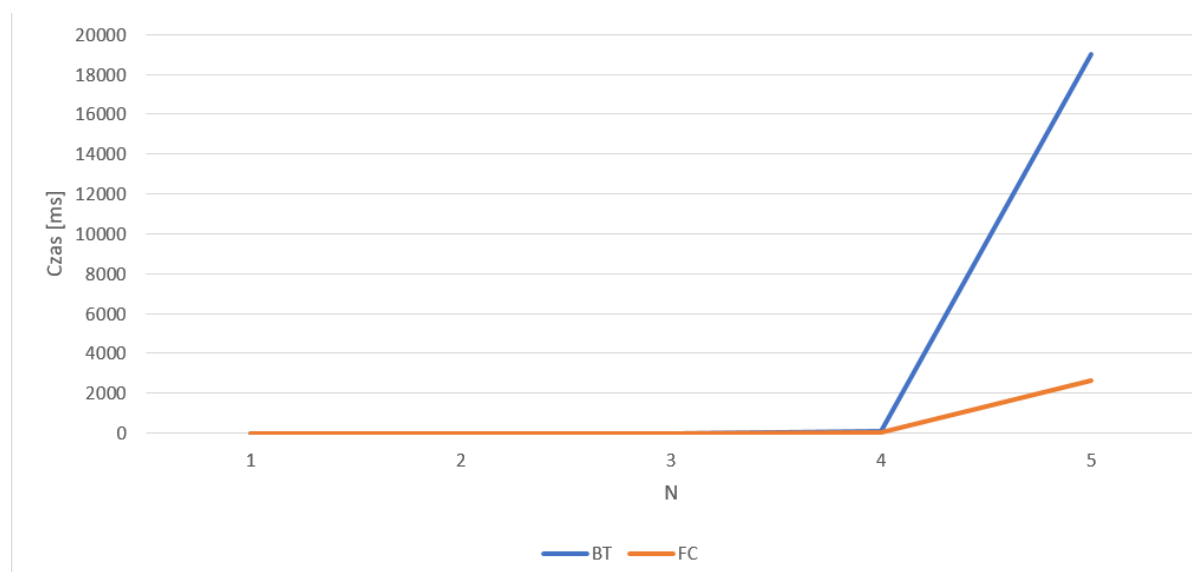
N	1 – 3	4	5
BT	0	85	19019
FC	0	16	2645

Zauważono, że **przeszukiwanie przyrostowe** jest szybszym sposobem na znalezienie wszystkich rozwiązań. **Sprawdzanie w przód** notuje wielokrotnie gorsze czasy.

Zależność liczby odwiedzonych węzłów przez algorytm, od wielkości problemu, dla Kwadratu łacińskiego.

N	1	2	3	4	5
BT	2	15	247	20421	10764431
FC	2	9	94	5585	2178805

Zauważono, że **algorytm sprawdzania w przód** odwiedza dużo więcej węzłów, niż **algorytm przyrostowy**. Różnica jest nawet kilkukrotna dla badanych danych.



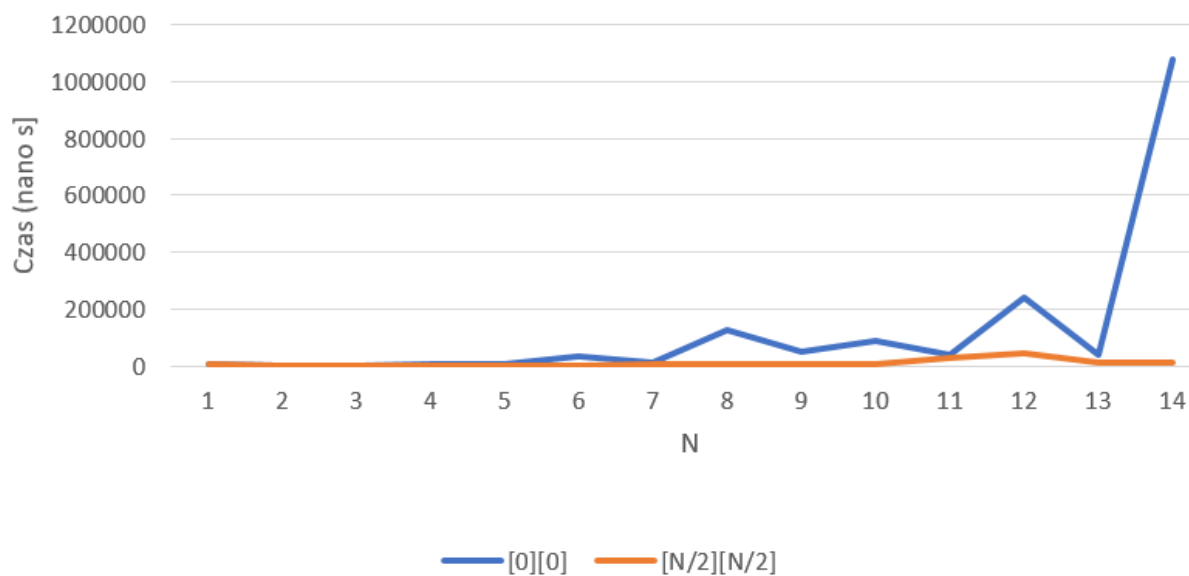
## 5. Heurystyka

### 5.1. Punkt startu

Pierwsze podejście to miana punktu startowego algorytmu BT dla N-hetmanów z lewego górnego rogu, na punkt  $[N/2][N/2]$ , dla znalezienia pierwszego rozwiązania.

Porównanie czasów (nanosekundy) dla wywołań lewy górny róg oraz środek.

N	[0][0]	$[N/2][N/2]$
1	6694	6694
2	2677	1785
3	4016	2677
4	5801	2232
5	6693	4462
6	37038	4017
7	10709	6694
8	128072	6693
9	48641	8033
10	86572	6247
11	39269	27667
12	241866	44625
13	42394	13388
14	1080363	13834



## 5.2. Kontrola kolumn i wierszy

Zastosowano rozwiązanie, gdzie przy wstawianiu hetmana, zajmowany jest cały wiersz oraz kolumna w którym się znajduje. Pozwoliło to usprawnić sprawdzanie możliwości wstawienia dla kolejnego hetmana oraz znacznie przyspieszyło działanie algorytmu.

```
for (int i = 0; i < column; i++) {
    if (board[row][i]) {
        return false;
    }
}

ZAMIENIONO NA

if(columnControl[row]){ //column controll
    return false;
}
if(rowControl[column]){ //row controll
    return false;
}
```

iteracje:

===== N QUEENS =====

### BACKTRACKING

n = 14 Liczba rozwiązań: 365596  
uruchomień: 8224.0ms

Wywołań rekursywnych: 27358553

Mediana z 10

### FORWARD CHECKING

n = 14 Liczba rozwiązań: 365596  
uruchomień: 7854.0ms

Wywołań rekursywnych: 17693457

Mediana z 10

kontrola kolumn i wierszy:

===== N QUEENS =====

### BACKTRACKING

n = 14 Liczba rozwiązań: 365596  
uruchomień: 5294.0ms

Wywołań rekursywnych: 27358553

Mediana z 10

### FORWARD CHECKING

n = 14 Liczba rozwiązań: 365596  
uruchomień: 4668.0ms

Wywołań rekursywnych: 17693457

Mediana z 10



## 6. Podsumowanie

Algorytm sprawdzania wprzód i sprawdzania przyrostowego są bardzo ciekawymi sposobami rozwiązania przedstawionego zadania. Algorytmy znacznie różnią się od siebie.

Dla  $N$ -królowych, algorytm przyrostowy był szybszy jedynie do pewnego momentu. Po przekroczeniu  $N=10$ , jego czas rósł bardzo szybko, a algorytm sprawdzania wprzód zachowywał wysoką efektywność, przy nie aż tak rosnącym czasie. (Wykres został przedstawiony w treści zadania.)

Na działanie algorytmów wpływa ich implementacja oraz to jak bardzo panujemy nad kodem. Analizując dane z *profilera*, można było znaleźć miejsce gdzie złożoność działań była wysoka i usprawnić działanie algorytmu zyskując kolejne części sekundy. Poznanie narzędzia jakim jest *profiler* może pozwolić na pisanie bardziej efektywnego kodu w przyszłości.

Dobór odpowiednich heurystyk dla algorytmów może pomóc w optymalizacji działania programu przez ograniczenie węzłów jakie odwiedza algorytm.