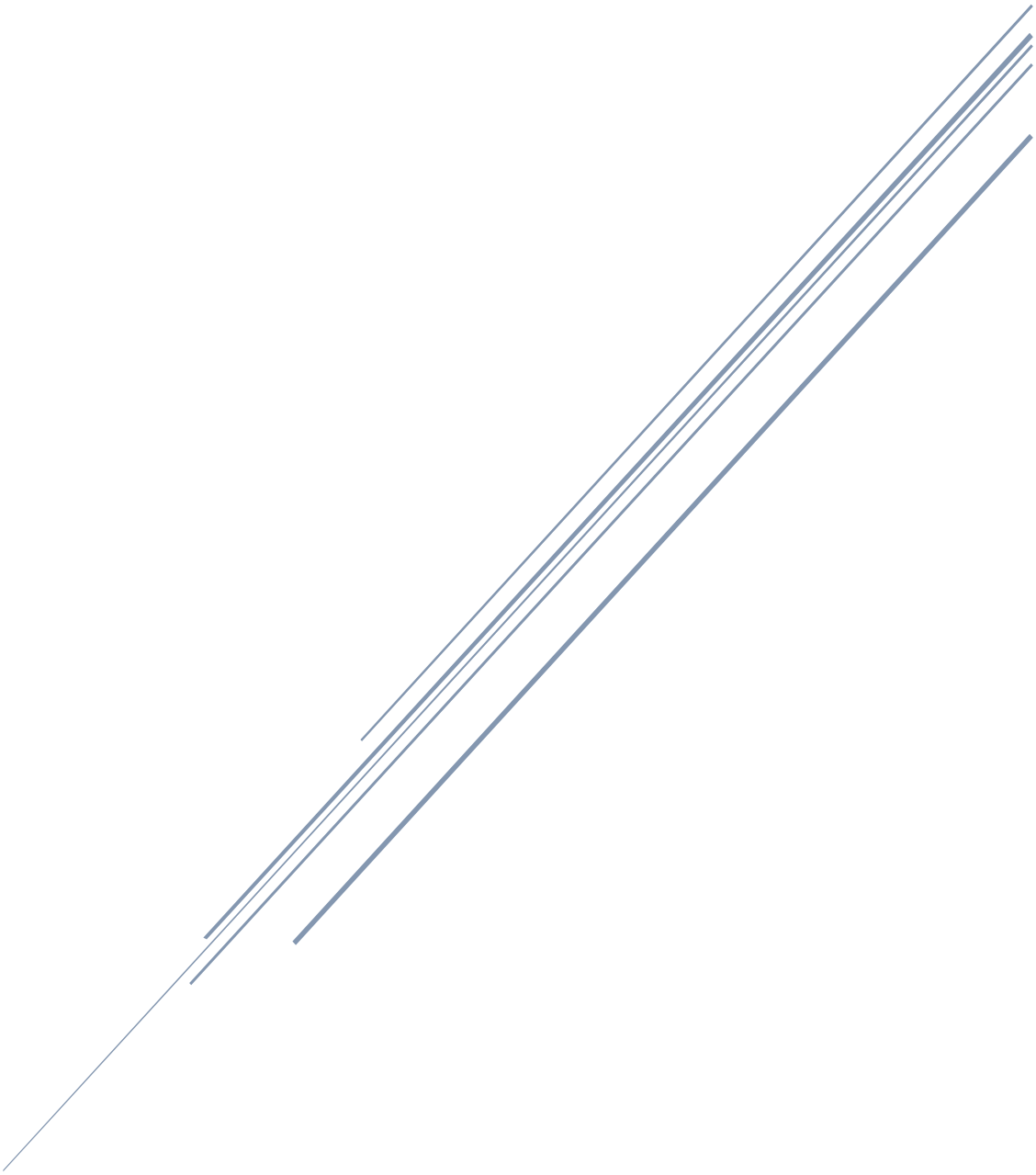
SZTUCZNA INTELIGENCJA I INŻ. WIEDZY

Constraint satisfaction problem



Michał Kanak
206906

Spis treści

Wstęp 2
1. CSP 2
2. Back tracking 2
3. Forward checking 2
4. Cel ćwiczenia 2
Definiowanie problemów 2
1. Sudoku (kwadrat łaciński) 2
2. Problem N-hetmanów 2
Sposoby rozwiązania 3
1. Algorytmy podstawowe 3
- N-hetmanów 3
-- backtracking 3
-- forward checking 3
- Sudoku 3
-- backtracking 3
-- forward checking 3
Wnioski 4
1. Profilowanie algorytmów 4
- Sudoku backtracking 4
- Sudoku forward checking 4
- Sudoku forward checking + 3 heurystyki (minimalne pozostałe wartości dla pól(one with least remaining values), najbardziej ograniczona zmienna(most constraining variable (variable with highest degree)) i pola z najmniejszą ilością wartości do uzupełniania(squares with the minimum remaining values)) 5
- N-hetmanów backtracking (dla 8 hetmanów) 6
- N-hetmanów backtracking (dla 12 hetmanów): 7
- N-hetmanów forward checking (dla 8 hetmanów) 8
- N-hetmanów forward checking (dla 12 hetmanów): 9
Wnioski 9

Wstęp

1. CSP

Constraint satisfaction problems – problem spełniania ograniczeń – jest to matematyczne pytanie zdefiniowane jak zbiór obiektów, których stan musi być spełniany przez pewną liczbę ograniczeń. Posiada liczbe skończonych ograniczeń w stosunku do zmiennych. Jest rozwiązywany za pomocą odpowiednich metod zdefiniowanych przez lata pracy nad tego typu problemami.

2. Back tracking

Algorytm z nawrotami – ogólny algorytm wyszukiwania wszystkich (lub kilku) rozwiązań niektórych problemów obliczeniowych, który stopniowo generuje kandydatów na rozwiązanie jednak, gdy stwierdzi, że znaleziony kandydat *c* nie może być poprawnym rozwiązaniem, nawraca (ang. "backtracks") do punktu, gdzie może podjąć inną decyzje związaną z jego budową.

3. Forward checking

Algorytm sprawdzania wprzód - połączenie przeszukiwania z nawracaniem ze sprawdzaniem spójności więzów. Przeszukiwanie sprawdza, czy dotychczas przypisane zmienne nie eliminują wszystkich wartości dla którejś z nieprzypisanych zmiennych.
Cofa się, jeśli któraś zmienna nie ma już żadnej dopuszczalnej wartości.

4. Cel ćwiczenia

Rozwiązanie dwóch wybranych problemów spełniania ograniczeń, za pomocą dwóch wyżej wymienionych metod. Sprofilowanie swoich rozwiązań pod względem wydajności, porównanie ich, wyciągnięcie wniosków i naniesienie wyników na sprawozdanie. W moim przypadku będą to problemy: „Sudoku” oraz „N-Hetmanów”.

Definiowanie problemów

1. Sudoku (kwadrat łaciński)

Sformułowanie problemu:

Variables: **9 bloków(3x3) = 81 pól**
Domain: **D = {1..9}**,
Constraints: 1. W poziomie i pionie cyfra można wystąpić tylko raz w wymiarze całego diagramu.
2. W danym bloku cyfra może wystąpić tylko raz

Problem, którego celem jest wypełnienie diagramu 9 × 9 w taki sposób, aby w każdym wierszu, w każdej kolumnie i w każdym z dziewięciu pogrubionych kwadratów 3 × 3 (zwanych „blokami”) znalazło się po jednej cyfrze od 1 do 9. Rozwiązywanie ręcznie sprowadza się do tego że do diagramu - cyfry należy wpisywać jedynie w miejsca, gdzie na pewno powinny się znaleźć. Niepewne miejsca można tylko zanotować lub zaznaczyć, by uniknąć kreślenia i poprawek. W podejściu CSP, będziemy uzupełniać pola kolejno, rozwiązując wszystkie możliwości, w celu usprawnienia można stosować heurystyki.

2. Problem N-hetmanów

Sformułowanie problemu dla 8 hetmanów:
Variables: **8 wierszy na planszy**
Domain: **D = {1..n} dla pozycji w wierszu**
Constraints: Hetmany muszą być ustawione na planszy w taki sposób by każdy z nich nie był w jednej linii pionowo, poziomo lub pod ukosem z innym hetmanem.
$x_i <> x_j, x_i - x_j <> i - j, x_j - x_i <> i - j$

Problem podstawowy (dla 8 hetmanów) jest pytaniem w jaki sposób rozstawić osiem hetmanów na tradycyjnej szachownicy 8x8 tak, aby wzajemnie się nie atakowały? Ile jest możliwych rozstawień? Przez rozstawienie podstawowe bądź rozwiązanie podstawowe należy

rozumieć rozwiązanie z uwzględnieniem wszystkich pokrewnych pozycji wynikających z odbić zwierciadlanych i obrotów.

Tabela przedstawia liczby możliwych ustawień dla szachownic o innym rozmiarze. Liczba rozwiązań rośnie w przybliżeniu wykładniczo, nie jest wyznaczony wzór na liczbę rozwiązań.

n	1	2	3	4	5	6	7	8	9	10	11	12
rozwiązań podstawowych	1	0	0	1	2	1	6	12	46	92	341	1.787
rozwiązań wszystkich	1	0	0	2	10	4	40	92	352	724	2.680	14.200

Sposoby rozwiązania

1. Algorytmy podstawowe

- N-hetmanów

-- backtracking

1) Zacznij od górnego lewego rogu
2) Jeśli wszyscy hetmani zostali przypisani return true
3) Dla wszystkich wierszy w danej kolumnie.:

    a) Jeśli hetman może zostać umieszczony na danym polu, oznacz pole i rekursywnie sprawdź czy stawiając w tym polu hetmana zbliżamy się do rozwiązania.
    b) Jeśli tak, return true.
    c) Jeśli nie, oznacz następne możliwe pole, odznaczając to błędnie zaznaczone.

4) Jeśli żaden z wierszy nie prowadzi do rozwiązania wywołaj backtracking.

-- forward checking

1) Sprawdzamy które rzędy nadają się na umieszczenie hetmana
2) Ustawiamy go w pierwszej możliwej pozycji,
3) Sprawdzamy domenę dla innych zmiennych,
4) Jeśli domena nie jest wyczerpana posuwamy się naprzód,
   w innym przypadku przerywamy wyszukiwanie

- Sudoku

-- backtracking

1. Znajdź nieprzypisany wiersz i kolumnę, jeśli taki nie istnieje - zwróć true
2. Dla cyfr od 1 do 9

    a) Jeśli nie istnieje konflikt w wierszu i kolumnie, przypisz cyfrę i rekursywnie spróbuj wypełnić resztę macierzy
    b) Jeśli się udało wypełnić wszystko, zwróć true
    c) W innym przypadku usuń liczbę i przypisz inną, a jeśli nie ma już innej liczby do przypisania – zwróć false

-- forward checking

1. Znajdź nieprzypisany wiersz i kolumnę, jeśli taki nie istnieje - zwróć true
2. Dla cyfr od 1 do 9

    a) jeśli wszystkie puste pola zostały sprawdzone każdą z cyfr – rozwiązanie nie istnieje
    b) jeśli nie ma już pustych pól sprawdź i zwróć wynik
    c) jeśli istnieją puste pola to idź do kolejnego
    d) jeśli możemy przypisać wartość, robimy to, jeśli nie backtracking

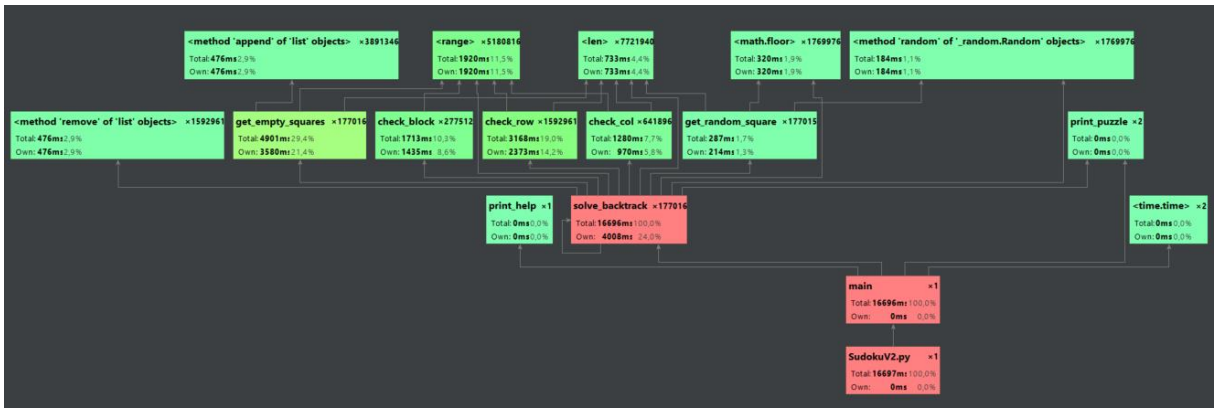
Wnioski

1. Profilowanie algorytmów

- Sudoku backtracking

Sudoku rozwiązywane w przykładach:	Poprawne rozwiązanie:
[0, 3, 0, 0, 8, 0, 0, 0, 6] [5, 0, 0, 2, 9, 4, 7, 1, 0] [0, 0, 0, 3, 0, 0, 5, 0, 0] [0, 0, 5, 0, 1, 0, 8, 0, 4] [4, 2, 0, 8, 0, 5, 0, 3, 9] [1, 0, 8, 0, 3, 0, 6, 0, 0] [0, 0, 3, 0, 0, 7, 0, 0, 0] [0, 4, 1, 6, 5, 3, 0, 0, 2] [2, 0, 0, 0, 4, 0, 0, 6, 0]	[7, 3, 4, 5, 8, 1, 2, 9, 6] [5, 8, 6, 2, 9, 4, 7, 1, 3] [9, 1, 2, 3, 7, 6, 5, 4, 8] [3, 6, 5, 7, 1, 9, 8, 2, 4] [4, 2, 7, 8, 6, 5, 1, 3, 9] [1, 9, 8, 4, 3, 2, 6, 5, 7] [6, 5, 3, 9, 2, 7, 4, 8, 1] [8, 4, 1, 6, 5, 3, 9, 7, 2] [2, 7, 9, 1, 4, 8, 3, 6, 5]

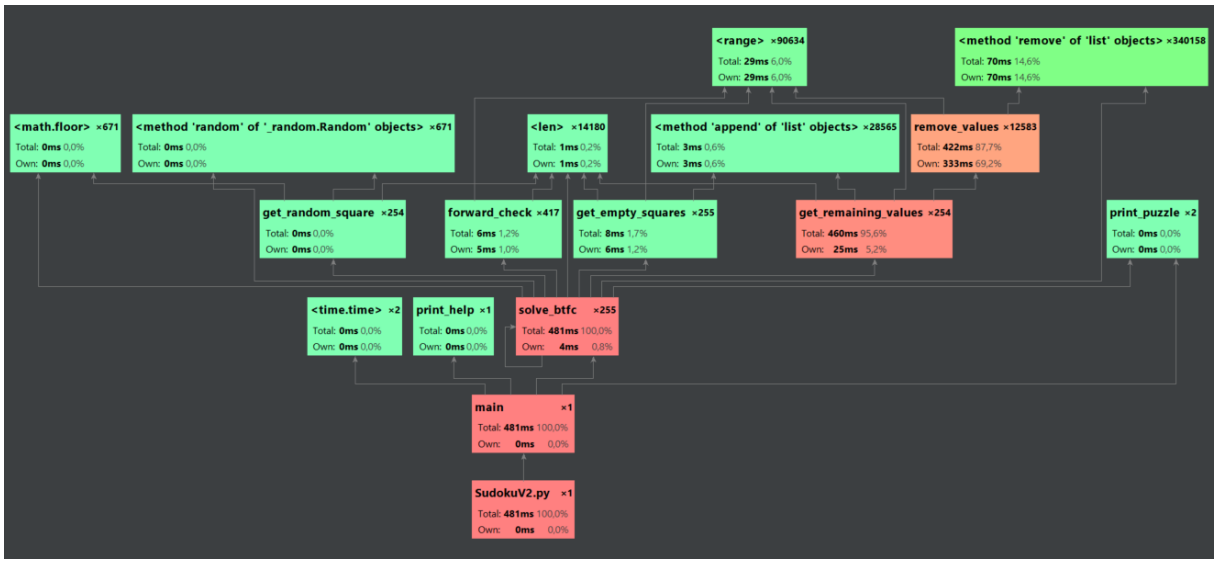
Name	Call Count	Time (ms)	Own Time (ms)
SudokuV2.py	1	16697 100.0%	0 0.0%
main	1	16696 100.0%	0 0.0%
solve_backtrack	177016	16696 100.0%	4008 24.0%
get_empty_squares	177016	4901 29.4%	3580 21.4%
check_row	1592961	3168 19.0%	2373 14.2%
<range>	5180816	1920 11.5%	1920 11.5%
check_block	277512	1713 10.3%	1435 8.6%
check_col	641896	1280 7.7%	970 5.8%
<len>	7721940	733 4.4%	733 4.4%
<method 'remove' of 'list' objects>	1592961	476 2.9%	476 2.9%
<method 'append' of 'list' objects>	3891346	476 2.9%	476 2.9%
<math.floor>	1769976	320 1.9%	320 1.9%
get_random_square	177015	287 1.7%	214 1.3%
<method 'random' of '_random.Random' objects>	1769976	184 1.1%	184 1.1%
print_puzzle	2	0 0.0%	0 0.0%
<time.time>	2	0 0.0%	0 0.0%



Poprzez ogromną ilość sprawdzeń kolejnych rozwiązań, przypisań wartości, później ich usuwanie oraz sprawdzeń poprawności w dwóch wymiarach (kolumn oraz wierszy) algorytm jest bardzo ciężki. Za każdym razem podczas backtrackingu zaczynamy praktycznie od nowa (usuwane są dobrane do tej pory wartości i zastępowane kolejnymi)

- Sudoku forward checking

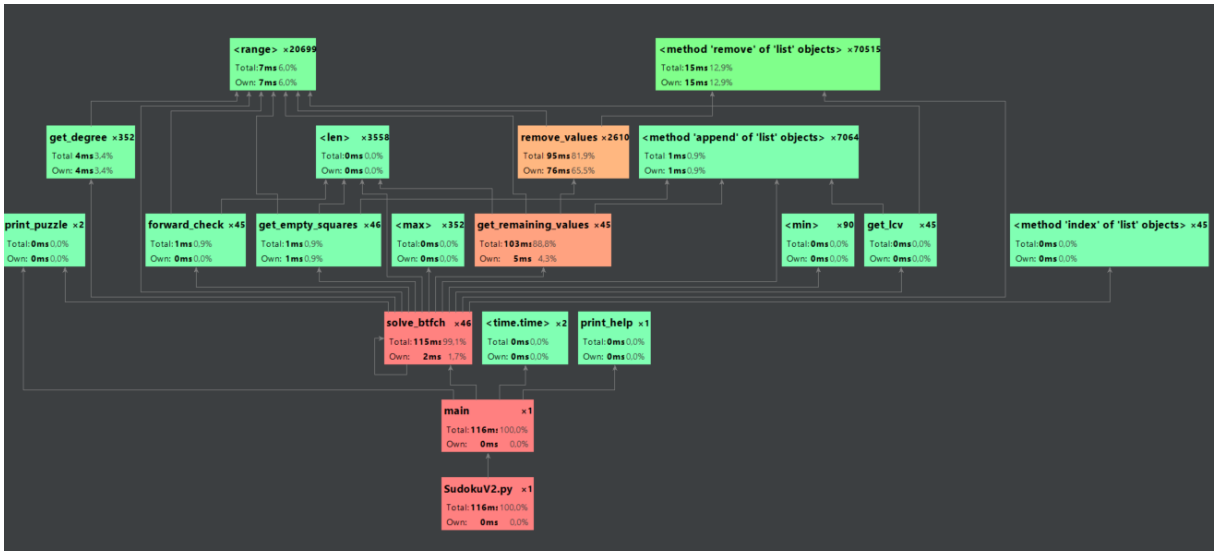
Name	Call Count	Time (ms)	Own Time (ms)
SudokuV2.py	1	481 100.0%	0 0.0%
solve_btfc	255	481 100.0%	4 0.8%
main	1	481 100.0%	0 0.0%
get_remaining_values	254	460 95.6%	25 5.2%
remove_values	12583	422 87.7%	333 69.2%
<method 'remove' of 'list' objects>	340158	70 14.6%	70 14.6%
<range>	90634	29 6.0%	29 6.0%
get_empty_squares	255	8 1.7%	6 1.2%
forward_check	417	6 1.2%	5 1.0%
<method 'append' of 'list' objects>	28565	3 0.6%	3 0.6%
<len>	14180	1 0.2%	1 0.2%
print_help	1	0 0.0%	0 0.0%
<method 'random' of '_random.Random' objects>	671	0 0.0%	0 0.0%
get_random_square	254	0 0.0%	0 0.0%
print_puzzle	2	0 0.0%	0 0.0%
<math.floor>	671	0 0.0%	0 0.0%
<time.time>	2	0 0.0%	0 0.0%



W tym przypadku sprawdzenia wprzód jest rozwiązaniem optymalnym w celu rozwiązania tego problemu. Unikając usuwania już dobranych wartości zyskujemy znacznie na szybkości, nawet pomimo ilości sprawdzeń i złożoności algorytm jest ponad 40 razy szybszy.

- Sudoku forward checking + 3 heurystyki (minimalne pozostałe wartości dla pól(one with least remaining values), najbardziej ograniczona zmienna(most constraining variable (variable with highest degree)) i pola z najmniejszą ilością wartości do uzupełniania(squares with the minimum remaining values))

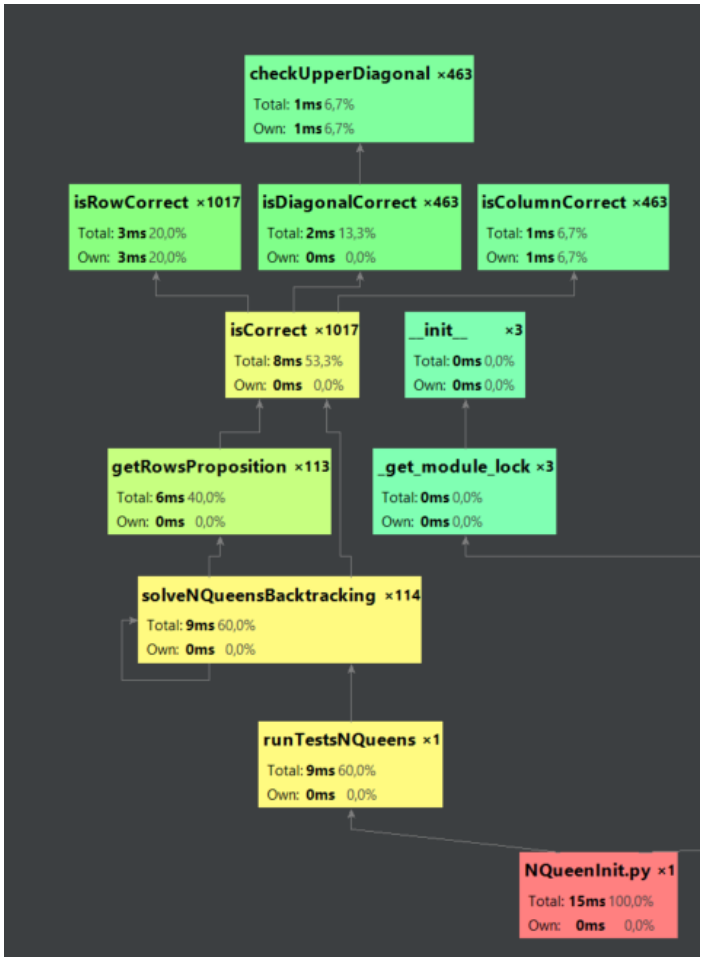
Name	Call Count	Time (ms)	Own Time (ms)
main	1	116 100,0%	0 0,0%
SudokuV2.py	1	116 100,0%	0 0,0%
solve_btfch	46	115 99,1%	2 1,7%
get_remaining_values	45	103 88,8%	5 4,3%
remove_values	2610	95 81,9%	76 65,5%
<method 'remove' of 'list' objects>	70515	15 12,9%	15 12,9%
<range>	20699	7 6,0%	7 6,0%
get_degree	352	4 3,4%	4 3,4%
forward_check	45	1 0,9%	0 0,0%
get_empty_squares	46	1 0,9%	1 0,9%
<method 'append' of 'list' objects>	7064	1 0,9%	1 0,9%
<time.time>	2	0 0,0%	0 0,0%
get_lcv	45	0 0,0%	0 0,0%
print_help	1	0 0,0%	0 0,0%
<min>	90	0 0,0%	0 0,0%
<len>	3558	0 0,0%	0 0,0%
<max>	352	0 0,0%	0 0,0%
<method 'index' of 'list' objects>	45	0 0,0%	0 0,0%
print_puzzle	2	0 0,0%	0 0,0%



Stosując odpowiednio dobrane heurystyki(bazując na doświadczeniach) uzyskujemy jeszcze lepsze wyniki, optymalizując liczbę nieoptymalnych operacji. Algorytm jest 160 razy szybszy od backtrackingu oraz 4 razy szybszy od zwykłego forward checkingu!

- N-hetmanów backtracking (dla 8 hetmanów)

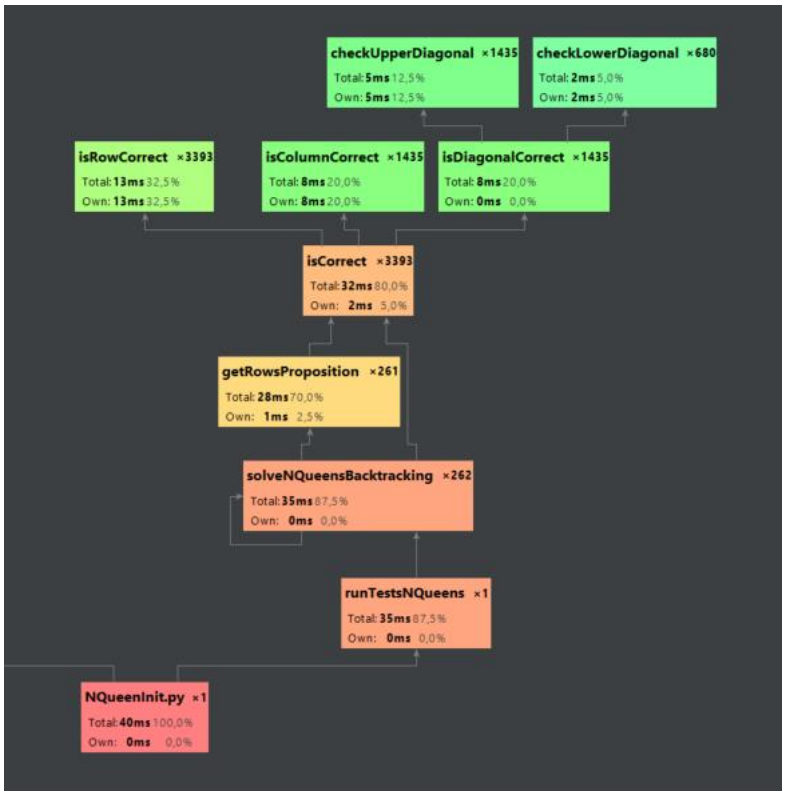
Name	Call Count	Time (ms)	Own Time (ms)
<built-in method builtins.len>	3171	0 0,0%	0 0,0%
isRowCorrect	1017	3 20,0%	3 20,0%
isCorrect	1017	8 53,3%	0 0,0%
isColumnCorrect	463	1 6,7%	1 6,7%
isDiagonalCorrect	463	2 13,3%	0 0,0%
checkUpperDiagonal	463	1 6,7%	1 6,7%
checkLowerDiagonal	286	0 0,0%	0 0,0%
<method 'append' of 'list' objects>	125	0 0,0%	0 0,0%
solveNQueensBacktracking	114	9 60,0%	0 0,0%
getRowsProposition	113	6 40,0%	0 0,0%
<method 'rstrip' of 'str' objects>	93	0 0,0%	0 0,0%



Metoda backtracking świetnie się sprawdza dla N-Hetmanów N=8, w krótkim czasie i za pomocą małej liczby nawrotów znajduje rozwiązanie. Jest metodą preferowaną do rozwiązania tego problemu.

- N-hetmanów backtracking (dla 12 hetmanów):

Name	Call Count	Time (ms)	Own Time (ms)
<built-in method builtins.len>	9642	0 0,0%	0 0,0%
isRowCorrect	3393	13 31,7%	13 31,7%
isCorrect	3393	33 80,5%	2 4,9%
isColumnCorrect	1435	8 19,5%	8 19,5%
isDiagonalCorrect	1435	9 22,0%	0 0,0%
checkUpperDiagonal	1435	5 12,2%	5 12,2%
checkLowerDiagonal	680	2 4,9%	2 4,9%
<method 'append' of 'list' objects>	296	0 0,0%	0 0,0%
solveNQueensBacktracking	262	36 87,8%	0 0,0%
getRowsProposition	261	28 68,3%	1 2,4%
recurser	157	0 0,0%	0 0,0%
__call__	144	0 0,0%	0 0,0%
_extendLine	144	0 0,0%	0 0,0%
<method 'rstrip' of 'str' objects>	97	0 0,0%	0 0,0%



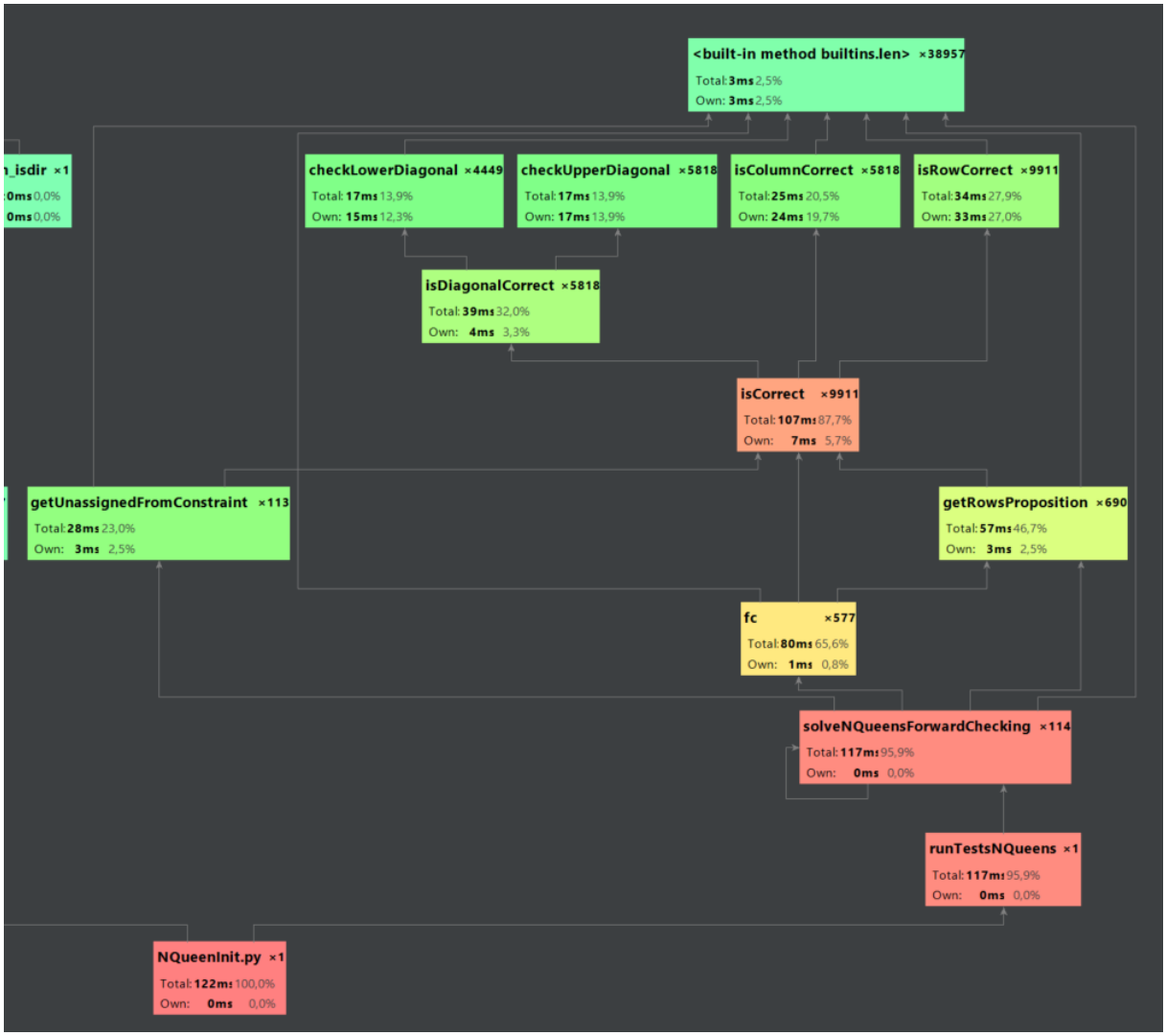
Wynik programu dla N == 12:

[[1 0 0 0 0 0 0 0 0 0 0 0]
[0 0 0 0 0 0 0 0 1 0 0 0]
[0 1 0 0 0 0 0 0 0 0 0 0]
[0 0 0 0 0 0 0 0 0 0 0 1]
[0 0 1 0 0 0 0 0 0 0 0 0]
[0 0 0 0 0 0 1 0 0 0 0 0]
[0 0 0 0 0 0 0 0 0 1 0 0]
[0 0 0 1 0 0 0 0 0 0 0 0]
[0 0 0 0 0 0 0 0 0 0 1 0]
[0 0 0 0 1 0 0 0 0 0 0 0]
[0 0 0 0 0 0 0 1 0 0 0 0]
[0 0 0 0 0 1 0 0 0 0 0 0]]
Solved in time = 0.6444778442382812

Jak widać czas rozwiązania metodą backtracking w tym przypadku nie rośnie tak znacząco pomimo znacznego zwiększenia zakresu planszy.

- N-hetmanów forward checking (dla 8 hetmanów)

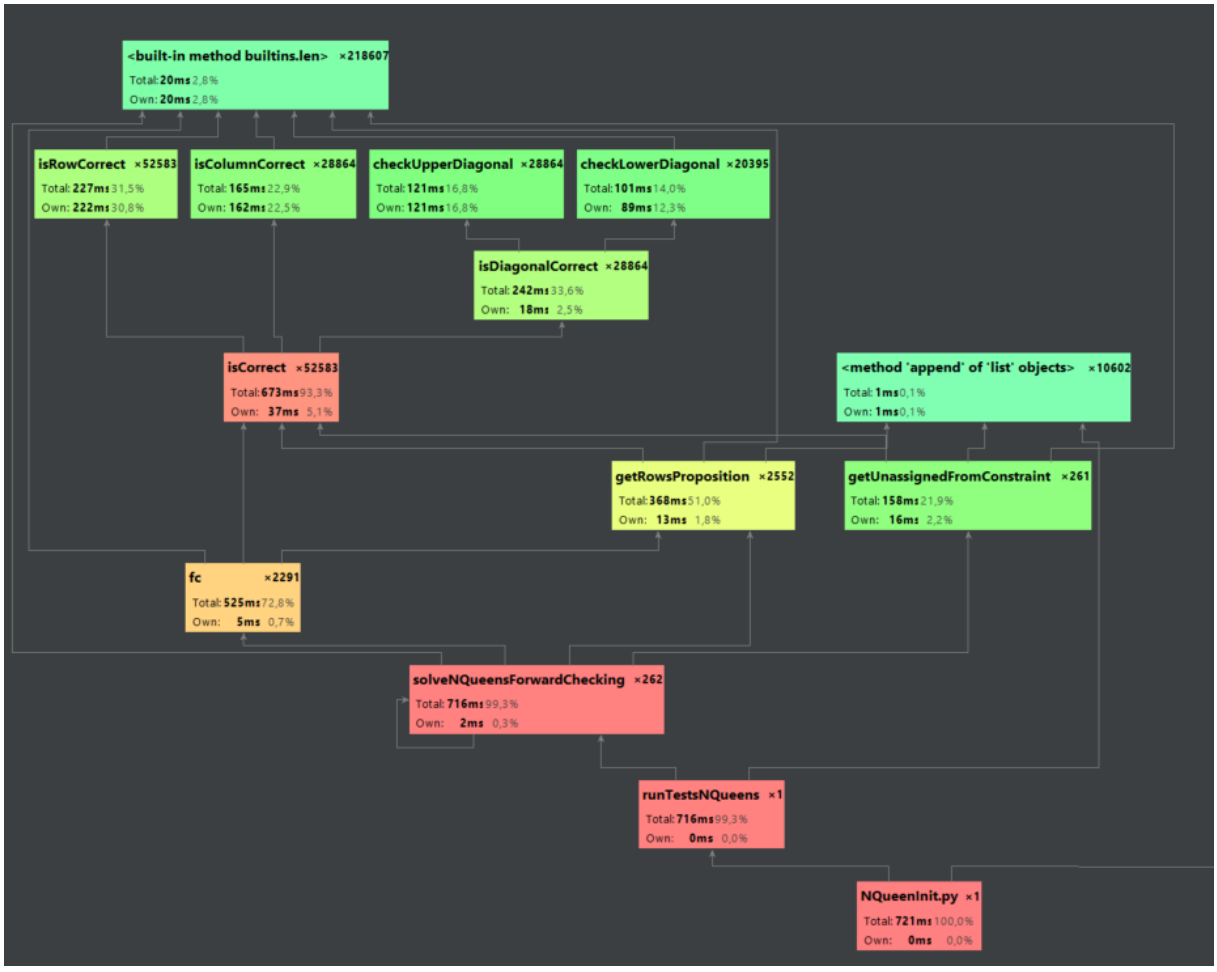
Name	Call Count	Time (ms)	Own Time (ms)
<built-in method builtins.len>	38957	3 2,5%	3 2,5%
isRowCorrect	9911	33 27,7%	32 26,9%
isCorrect	9911	104 87,4%	7 5,9%
isColumnCorrect	5818	24 20,2%	24 20,2%
isDiagonalCorrect	5818	38 31,9%	3 2,5%
checkUpperDiagonal	5818	17 14,3%	17 14,3%
checkLowerDiagonal	4449	16 13,4%	14 11,8%
<method 'append' of 'list' objects>	2357	0 0,0%	0 0,0%
getRowsProposition	690	55 46,2%	2 1,7%
fc	577	78 65,5%	1 0,8%
__init__	577	0 0,0%	0 0,0%
solveNQueensForwardChecking	114	114 95,8%	0 0,0%
getUnassignedFromConstraint	113	27 22,7%	3 2,5%
<method 'rstrip' of 'str' objects>	93	0 0,0%	0 0,0%
recurser	73	0 0,0%	0 0,0%
__call__	64	0 0,0%	0 0,0%
_extendLine	64	0 0,0%	0 0,0%
<method 'join' of 'str' objects>	44	0 0,0%	0 0,0%



Forward checking nie jest tak wydajny dla rozwiązywania tego problemu, jak backtracking, wpływa na to ilość operacji forward check oraz sprawdzeń „wygenerowanych” konfiguracji.

- N-hetmanów forward checking (dla 12 hetmanów):

Name	Call Count	Time (ms)	Own Time (ms)
<built-in method builtins.len>	218607	20 2,8%	20 2,8%
isRowCorrect	52583	227 31,5%	222 30,8%
isCorrect	52583	673 93,3%	37 5,1%
isColumnCorrect	28864	165 22,9%	162 22,5%
isDiagonalCorrect	28864	242 33,6%	18 2,5%
checkUpperDiagonal	28864	121 16,8%	121 16,8%
checkLowerDiagonal	20395	101 14,0%	89 12,3%
<method 'append' of 'list' objects>	10602	1 0,1%	1 0,1%
getRowsProposition	2552	368 51,0%	13 1,8%
fc	2291	525 72,8%	5 0,7%
__init__	2291	0 0,0%	0 0,0%
solveNQueensForwardChecking	262	716 99,3%	2 0,3%
getUnassignedFromConstraint	261	158 21,9%	16 2,2%
recurser	157	0 0,0%	0 0,0%
__call__	144	0 0,0%	0 0,0%
_extendLine	144	0 0,0%	0 0,0%
<method 'rstrip' of 'str' objects>	97	0 0,0%	0 0,0%
<method 'join' of 'str' objects>	44	0 0,0%	0 0,0%



Wnioski

Jak widzimy z powyższych statystyk profiler'a, rozwiązywanie problemów CSP może ponosić koszty ogromnej liczby operacji oraz czasu, by rozwiązać problem. Należy także pamiętać, o złożoności operacyjnej rosnącej wykładniczo w przypadku problemu N-hetmanów. Wiele też zależy od sposobu realizacji algorytmu i unikania zbędnych operacji przy rekurencyjnie wykonywanych operacjach

Backtracking vs forward checking

Nie ma uniwersalnej odpowiedzi na to jaką metodą rozwiązywać wszystkie problemy CSP. Metoda forward checking lepiej sprawdza się dla problemu sudoku, a o wiele gorzej wypada od backtrackingu w przypadku problemu N-hetmanów. Dla każdego problemu należy podejść indywidualnie i rozpatrzyć, które rozwiązanie będzie optymalniejsze.

Zastosowanie heurystyk, a wpływ na sprawność algorytmów

Jak widać na przykładzie sudoku, stosując odpowiednio dobrane heurystyki(bazując na doświadczeniach) uzyskujemy jeszcze lepsze wyniki, optymalizując liczbę nieoptymalnych operacji. Algorytm jest 160 razy szybszy od backtrackingu oraz 4 razy szybszy od zwykłego forward checkingu! W tym przypadku zastosowałem - minimalne pozostałe wartości, stopień i najmniejsza wartość ograniczająca, jako heurystyki zwiększające szybkość znajdowania rozwiązania, heurystyki zmniejszają również znacząco ilość operacji, które musi wykonać komputer w celu rozwiązania problemu.

W przypadku sudoku, możemy stosować takie heurystyki (oraz) funkcje optymalizujące:

Dla wyboru kolejnej komórki:
- komórki z lewej do prawej z góry do dołu
- komórki z prawej do lewej z dołu do góry
- losowo wybrane komórki
- najbliższa pusta komórka od środka diagramu
- komórka która ma najmniej możliwych rozwiązań
- komórka która ma najwięcej możliwych rozwiązań
- -||- najmniej pustych sąsiadów
- -||- najwięcej pustych sąsiadów
- -||- najmniej wypełnionych sąsiadów
- -||- najwięcej wypełnionych sąsiadów
- komórka której sąsiedzi mają najwięcej możliwych rozwiązań
- komórka której sąsiedzi mają najmniej możliwych rozwiązań

Dla wyboru kolejnej liczby w komórce:

- najmniejsza liczba
- największa liczba
- losowo wybrana liczba
- heurystycznie, najmniej używana liczba na całym diagramie
- heurystycznie, najbardziej używana liczba na całym diagramie
- heurystycznie, których zostało najmniej do uzupełnienia na całym diagramie
- heurystycznie, których zostało najwięcej do uzupełnienia na całym diagramie
- -||- najmniej/najwięcej do uzupełnienia w pobliżu danej komórki.