Problem N-hetmanów.

Problem polega na ustawieniu na tablicy w ten sposób, aby każdy z hetmanów nie mógł zaatakować innego. Hetman jest w stanie zaatakować innego hetmana, gdy figura znajduje się na przekątnej, bądź na prostej, na której znajduje się inny hetman.

Problem jest problemem CSP (spełniania ograniczeń). W podanym przypadku zmiennymi są położenia królowej na planszy (mogą być prezentowane przez np. numer wiersza w kolumnie, gdzie należy umieścić królową), dziedziną (hetman/wolne pole), oraz więzy, tj. spełnienie poniższych założeń:

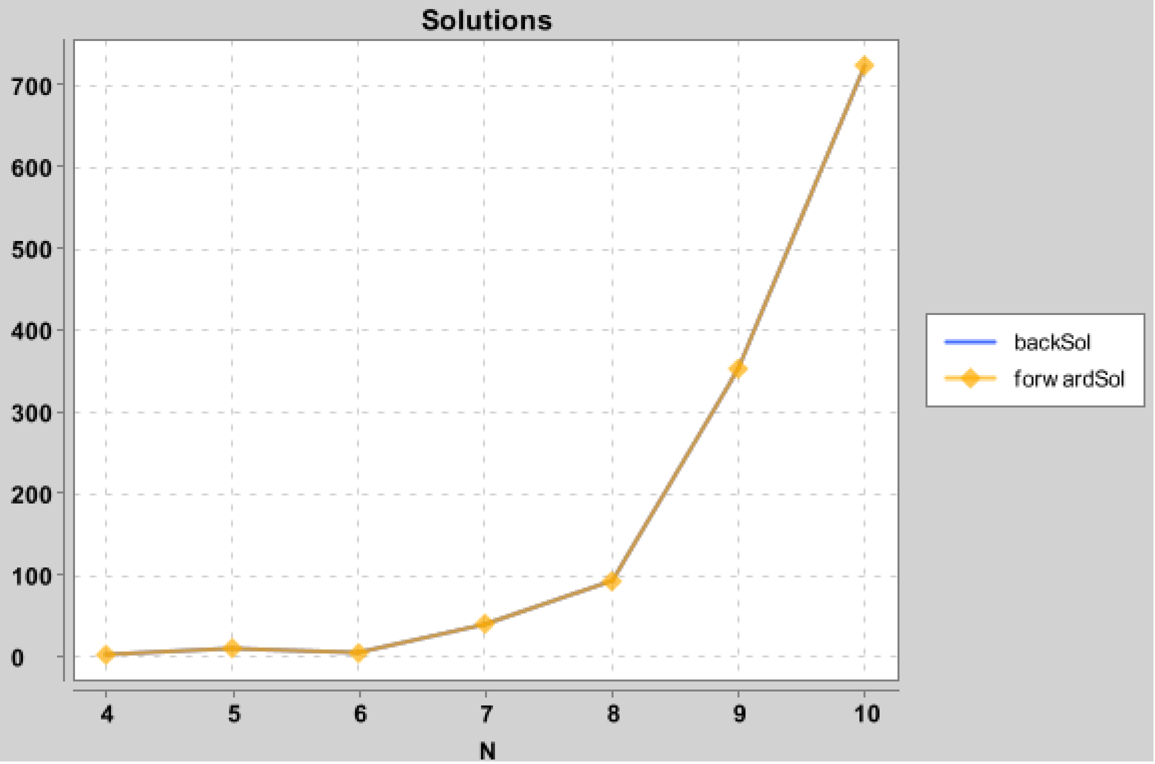
Niech (i,j) oraz (m,n) oznaczają współrzędne dwóch hetmanów. Dwa hetmany stoją na jednej linii wtedy i tylko wtedy i = m lub j = n. Dwa hetmany stoją przekątnej wtedy i tylko wtedy, gdy i = m±x oraz gdy j = n±x. Na jednej linii, przekątnej musi być dokładnie jeden hetman.

Problem można rozwiązać za pomocą wielu algorytmów, między innymi forward checking oraz backtracking.

Backtracking polega na tym, że algorytm szuka wszystkich rozwiązań, jednak podczas szukania kandydata, gdy stwierdzi, że ten nie może być rozwiązaniem nawraca do momentu, gdzie może podjąć inną budowę rozwiązania.

Forward checking różni się od niego, że zanim algorytm podejmie próbę budowania rozwiązania sprawdzi, czy jest ono możliwe.

Zarówno liczba rozwiązań jak i kombinacji nie jest liniowa dla zadanego problemu (dla przykładu – tablica 8x8 posiada 4 426 165 368 kombinacji, ale jedynie 92 rozwiązania). Dlatego oba algorytmy starają nakładać ograniczenia i sprawdzać tylko te potencjalne rozwiązania, które mogą przynieść pozytywny rezultat.



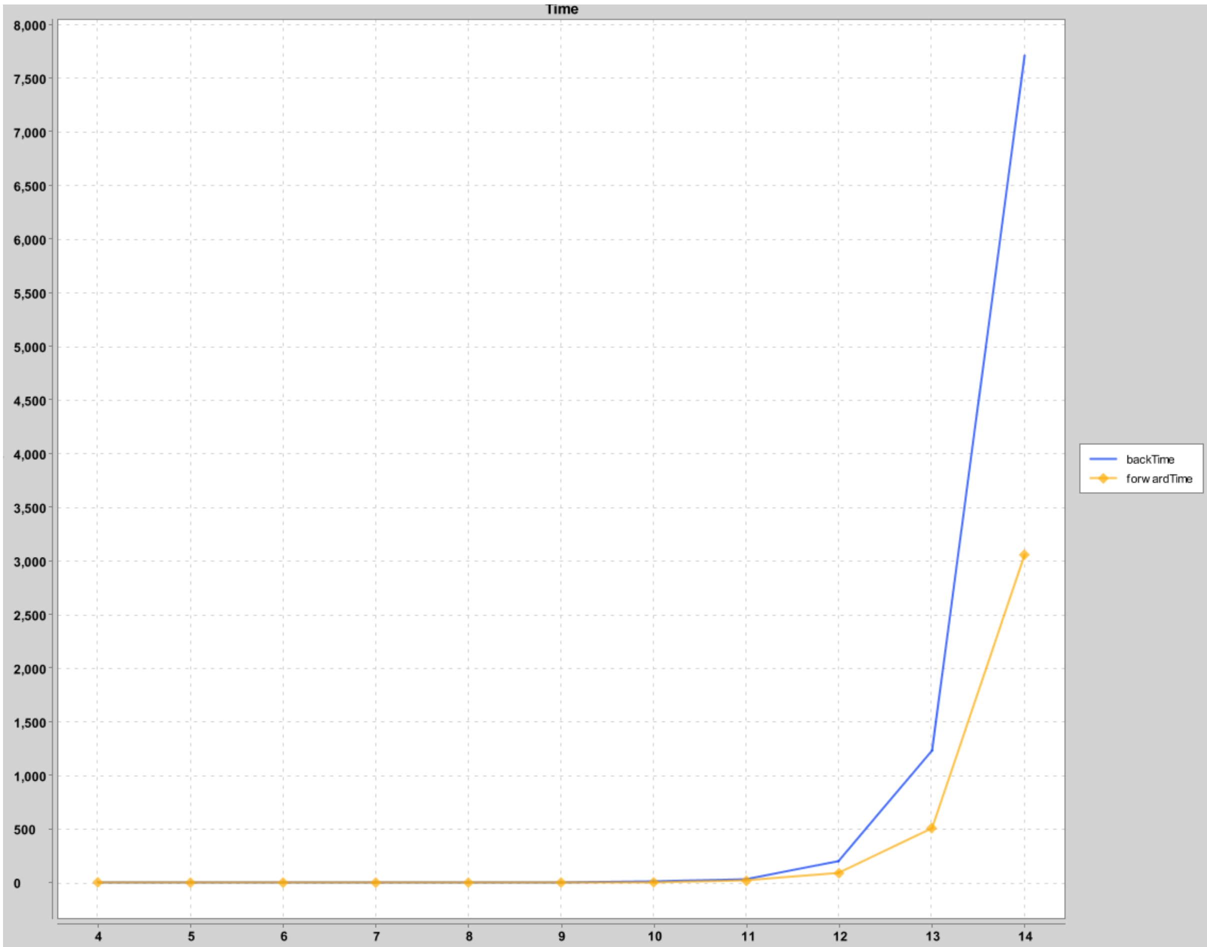
Algorytmy będą różnić się prawdopodobnie:

- Czas trwania algorytmu (z początkową przewagą backtrackingu)
- Liczbą rekurencyjną wywołań algorytmu (z początkową przewagą backtrackingu).

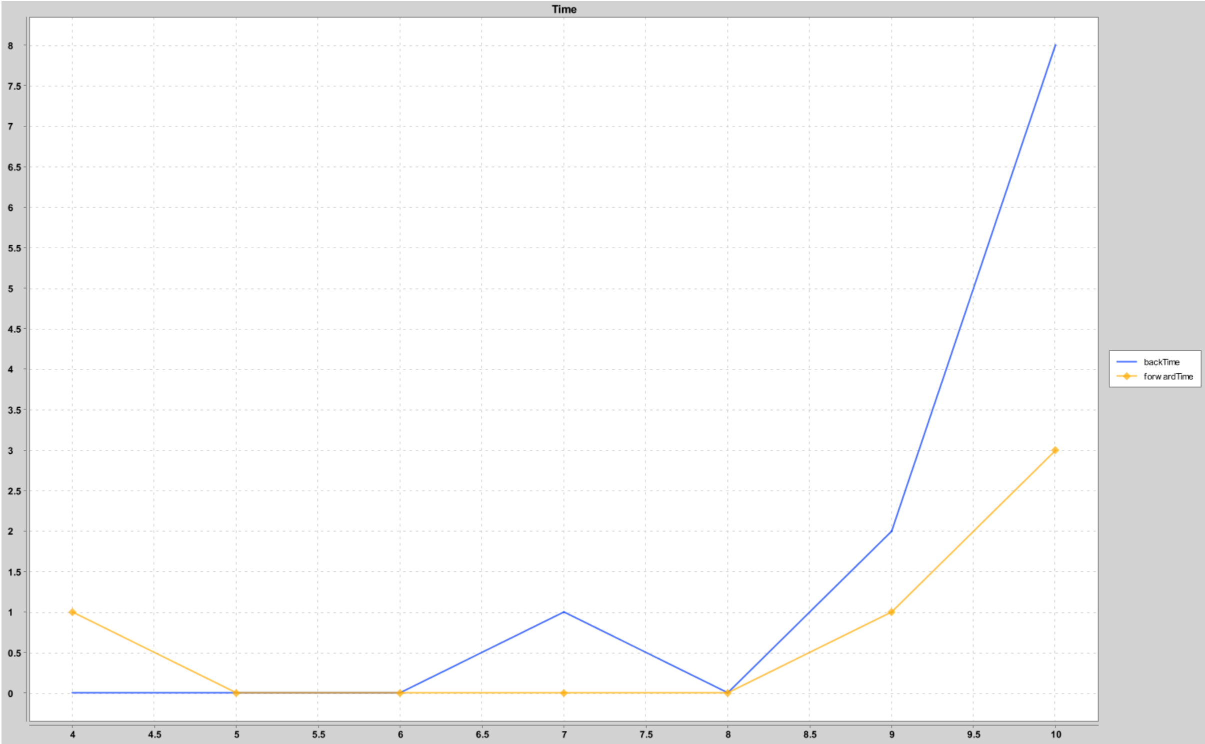
Program został zaimpementowany w Javie

Zbadano następujące dane:

1) Czas

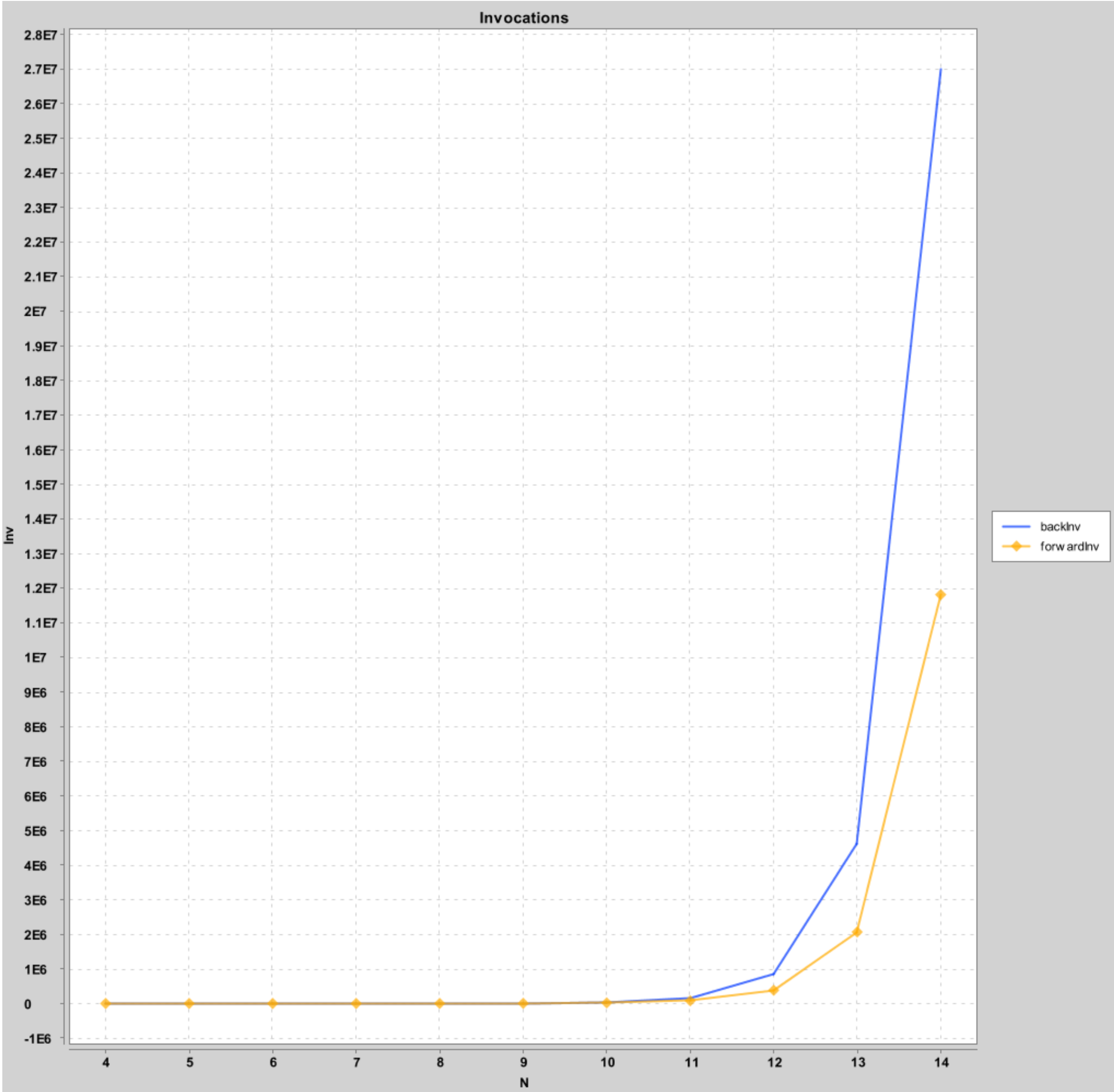


Jak widać, od pewnego poziomu forward checking radzi sobie zdecydowanie lepiej. Sprawdzenie, czy warto generować rozwiązanie i „wchodzić" do danego węzła sprawdza się dobrze, powodując, że dla większych N czas wykonania maleje nawet kilkukrotnie.
Aby zobaczyć skalę dla mniejszych N program uruchomiono dla N w zakresie od 4 do 10:



Jednak, z powodu bardzo małych różnic ciężko stwierdzić który z tych algorytmów radzi sobie lepiej dla N < 8. W teorii, powinien być to backtracking.

2) Liczba powrotów (wywołań metody rekurencyjnie)



Dla każdego przypadku liczba powrotów dla forward checkingu jest mniejsza. Oznacza to, że algorytm „mądrzej" wybiera rozwiązania, dzięki czemu przy większym N może on bardziej optymalnie rozwiązywać problem.

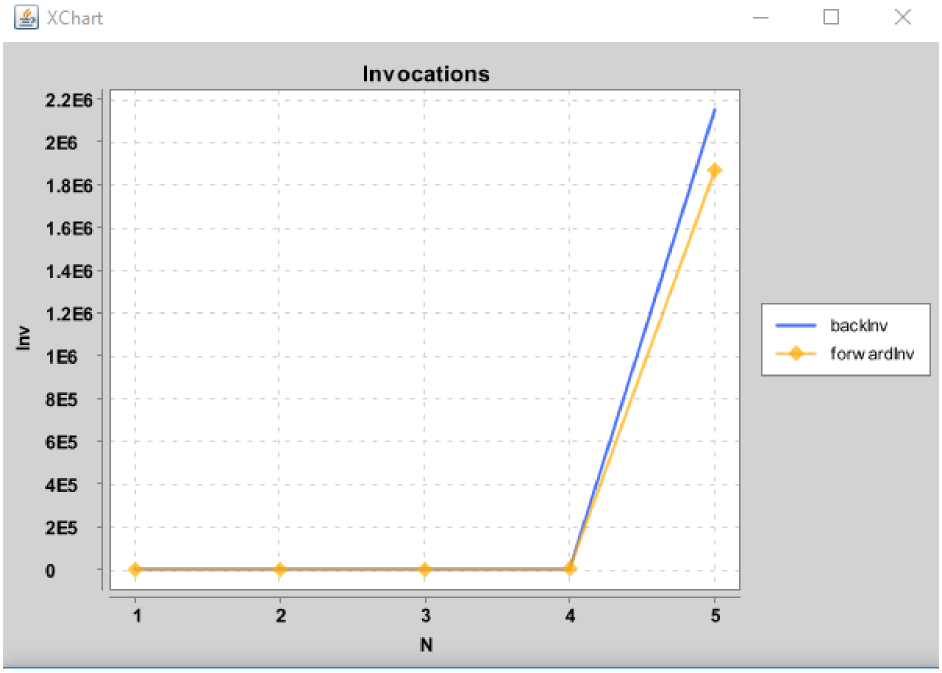
	Time		Inv			
N	BT	FC	BT	FC	time	inv
6	1	1	149	83	1	1.795180723
7	1	1	512	294	1	1.741496599
8	1	1	1965	1073	1	1.831314073
9	1	1	8042	4368	1	1.841117216
10	7	3	38415	16765	2.333333333	2.291380853
11	36	16	164246	76074	2.25	2.159029366
12	194	82	841989	383107	2.365853659	2.197790696

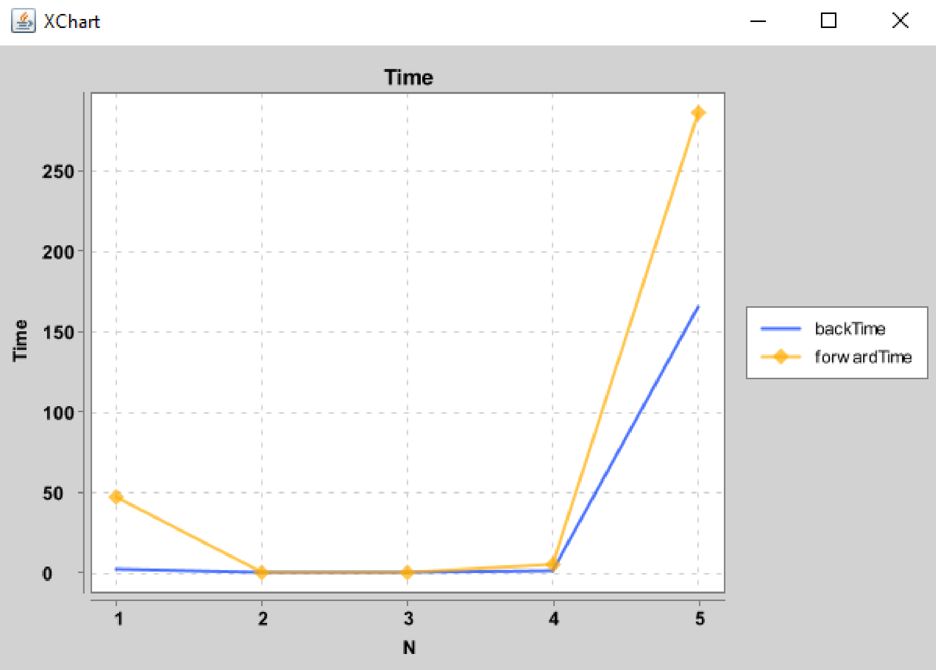
Jak widać czas dla bactrackingu jest coraz gorszy. Jednak stosunek ilości wywołań metody jest zmienna, i ciężko określić trend.

Latin Square

Latin Square jest kolejnym problemem CSP. W tym wypadku naszym zadaniem jest rozmieszczenie liczb z zakresu 0..N na macierzy o rozmiarze NxN, w ten sposób, aby każda podmacierz o rozmiarze 1x1 była wypełniona, oraz aby żadna z liczb nie powtarzała się w wierszu, lub komórce.

Sytuacja wygląda bardzo podobnie jak w przypadku problemu N-hetmanów. Liczba inwokacji w przypadku algorytmu forward checking jest mniejsza, lecz czas jest większy. Możemy jednak podejrzewać, że czas ten przy większych n będzie na korzyć Forward Checkingu.





	Time		Inv		
N	BT	FC	BT	FC	time
2	1	1	2	2	1
3	7	7	1	1	1
4	1	2	1	1	0.5
5	162	278	162	278	0.582733813
6	1245553	1939737	>Int.Max	>Int.Max	0.64212468
					Czas bactrackingu się wydłuża

Jak widzimy, czas bactrackingu będzie zbliżał się do forward checkingu, aż w końcu, przy większym kwadracie okaże się być prawdopodobnie lepszy. Spowodowane jest to mniejszą liczbą powrotów, pomimo większego czasu przebywania w danym polu.