RAPORT

Opis metody sprawdzania konfliktów dla zaimplementowanych reprezentacji:

1. **Tuple**

   board = [(0, 2), (1, 0), (2, 3), (3, 1)]

   Algorytm:

   - Porównuje każdą parę hetmanów sprawdzając wszystkie 3 typy konfliktów
   - Zwraca True przy pierwszym wykrytym konflikcie

   Problem: Przechowuje zbędne informacje, w prawidłowym rozwiązaniu każdy hetman musi być w innej kolumnie, więc jawne przechowywanie kolumn jest zbędne.

2. **Array**

   board = [[0,0,1,0], [1,0,0,0], [0,0,0,1], [0,1,0,0]]

   Algorytm:

   - Faza lokalizacji: Przeszukuje macierz $n \times n$ w celu znalezienia pozycji hetmanów (komórki z wartością 1)
   - Faza sprawdzenia: Stosuje identyczną logikę jak reprezentacja tuple

   Problem: Podwójne przetwarzanie danych, najpierw konwersja z macierzy do listy pozycji, potem sprawdzanie konfliktów. Marnotrawstwo pamięci dla przechowania informacji.

3. **Tuple**

   board = [2, 0, 3, 1] indeks=kolumna, wartość=wiersz

   Algorytm:

   - Indeks listy reprezentuje kolumnę hetmana, wartość - wiersz
   - Sprawdza tylko 2 warunki

   Zaleta: Inteligentna struktura danych.

4. **Tuple**

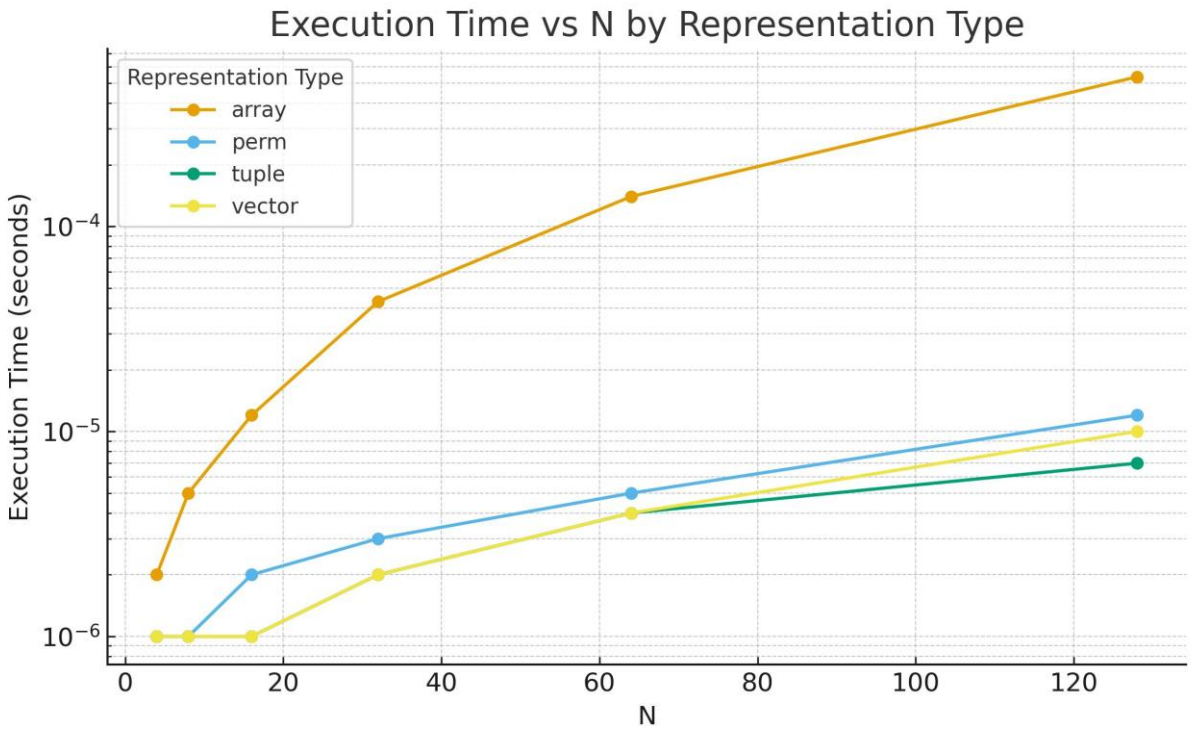
   board = (2, 0, 3, 1) permutacja 0..n-1

   Algorytm:

   - Sprawdza tylko 1 warunek, czyli przekątne
   - Kluczowa optymalizacja: konflikty wierszy i kolumn są strukturalnie niemożliwe

   Zaleta: Na pierwszy rzut oka najbardziej efektywne podejście.

N	representation_type	execution_time	conflict_rate
4	tuple	0.000001	1.00
4	array	0.000002	1.00
4	vector	0.000001	0.99
4	perm	0.000001	0.90
8	tuple	0.000001	1.00
8	array	0.000005	1.00
8	vector	0.000001	1.00
8	perm	0.000001	1.00
16	tuple	0.000001	1.00
16	array	0.000012	1.00
16	vector	0.000001	1.00
16	perm	0.000002	1.00
32	tuple	0.000002	1.00
32	array	0.000043	1.00
32	vector	0.000002	1.00
32	perm	0.000003	1.00
64	tuple	0.000004	1.00
64	array	0.000140	1.00
64	vector	0.000004	1.00
64	perm	0.000005	1.00
128	tuple	0.000007	1.00
128	array	0.000537	1.00
128	vector	0.000010	1.00
128	perm	0.000012	1.00



Wnioski końcowe

Czytelność kodu

- **Najbardziej intuicyjna:** array (naturalne odwzorowanie szachownicy)
- **Najbardziej bezpośrednia:** tuple (jawne współrzędne)
- **Najbardziej elegancka:** perm (minimalistyczny kod)
- **Kompromisowa:** vector (balans prostoty i efektywności)

Złożoność obliczeniowa i czas wykonania

- Tuple, Vector, Perm: $O(N^2)$ - iteracja przez wszystkie pary hetmanów
- Array: $O(N^2 + N^2) = O(N^2)$ - najpierw skanowanie tablicy (N^2), potem sprawdzanie par (N^2)

Praktyczny czas wykonania (z wyników)

Vector, Perm - najszybsze
Tuple - bardzo szybkie
Array – najwolniejszy

Intuicja + najlepszy algorytm do przeszukiwania

Tuple
Najbardziej uniwersalna reprezentacja. Algorytm może swobodnie dodawać i usuwać hetmany w dowolnych miejscach, ale musi samodzielnie pilnować wszystkich ograniczeń (wiersze, kolumny, przekątne).

Array
Intuicyjna wizualnie, łatwa do debugowania. Jednak bardzo nieefektywna, zajmuje dużo pamięci i jest wolna. W praktycznych algorytmach przeszukiwania lepiej jej unikać. Przydatna głównie do wizualizacji problemu.

Vector
Automatycznie zapewnia jednego hetmana na kolumnę, algorytm musi tylko sprawdzać wiersze i przekątne. Kompaktowa i szybka, najlepsza dla klasycznego przeszukiwania z głębokością.

Perm
Najlepsza dla algorytmów heurystycznych. Z definicji ma unikalnego hetmana w każdym wierszu i kolumnie, sprawdzamy tylko przekątne. Przestrzeń przeszukiwania drastycznie mniejsza (N! zamiast N^N). Operacje lokalne (swap dwóch elementów) są bardzo naturalne dla local search.