

Uniwersytet WSB Merito Warszawa

Tomasz Trawka

# ZASTOSOWANIE ALGORYTMÓW GENETYCZNYCH W ROZWIĄZYWANIU SUDOKU

PRACA PODYPLOMOWA  
Promotor: dr inż. Tomasz Długosz

Studia podyplomowe: Programowanie aplikacji Java

Wrocław, rok akademicki 2024/25

# Spis treści

<b>1. Wprowadzenie</b>	<b>2</b>
1.1. Dlaczego sudoku?	2
1.2. Jaki algorytm?	2
1.3. Dlaczego algorytmy genetyczne?	4
<b>2. Część teoretyczna</b>	<b>5</b>
2.1. Język programowania Java	5
2.2. Algorytmy genetyczne	6
2.2.1. Rys historyczny	6
2.2.2. Kluczowe pojęcia	7
2.2.3. Opis działania metody	8
2.2.4. Zalety i wady algorytmów genetycznych	11
2.2.5. Obszary zastosowania	12
<b>3. Część praktyczna</b>	<b>13</b>
3.1. Specyfikacja projektu	13
3.2. Specyfikacja elementów metody algorytmów genetycznych	13
3.2.1. Geny	14
3.2.2. Metody generowania pierwszego pokolenia	15
3.2.3. Metody obliczania funkcji dopasowania	15
3.2.4. Metody selekcji rodziców	16
3.2.5. Metody krzyżowania	16
3.3. Realizacja aplikacji	16
3.3.1. Podział kodu źródłowego	16
3.3.2. Przegląd głównych klas aplikacji	18
3.3.3. Interfejs graficzny aplikacji	24
<b>4. Podsumowanie projektu</b>	<b>x</b>
<b>5. Bibliografia</b>	<b>x</b>

# 1. Wprowadzenie

Sudoku jest stosunkowo młodą rozrywką logiczną. Została wynaleziona przez amerykańskiego Howarda Garnsa w 1979 w zupełnie odmiennej formie od tej, pod którą znamy ją obecnie [B01]. Dopiero rozpowszechnienie jej w Japonii w latach 80-tych XX wieku ukształtowało jej współczesną formę, pod którą jest znana na całym świecie. Olbrzymia popularność sudoku tłumaczona jest prostotą zasad oraz brakiem wymagań co do aparatu matematycznego koniecznego przy ich rozwiązywaniu. Wystarczy umiejętność logicznego myślenia i... cierpliwość.

## 1.1 Dlaczego sudoku?

Mimo prostoty zasad, sudoku jest stosunkowo trudnym tematem do rozwiązywania za pomocą metod algorytmicznych. Tradycyjne sudoku będące kwadratem o bokach 9 na 9 pól może przyjmować olbrzymie ilości rozwiązań. W 2006 roku dwaj matematycy - Bertram Felgenhauer i Frazer Jarvis - udowodnili w swojej pracy "Mathematics of Sudoku", że istnieje 6 670 903 752 021 072 936 960 różnych poprawnych plansz tego wariantu. Po odrzuceniu rozwiązań symetrycznych, powstałych przez 26 różnych wariantów odbić i obrotów, nadal pozostaje ponad  $5,47 \times 10^9$  unikalnych rozwiązań [A01].

Podczas rozwiązywania sudoku ułatwieniem jest wprowadzenie wartości prawidłowych dla niektórych pól. Jednak jeśli dla sudoku o rozmiarze 9 na 9 liczba wprowadzonych wartości jest mniejsza od 17, to jego rozwiązanie nie jest jednoznaczne. Ponadto, nawet dla układów z podanymi 17 cyframi nie wszystkie mają jednoznaczne rozwiązania - obecnie znanych jest ponad 49 tysięcy takich diagramów.

Obciążenie czasowe rozwiązywania sudoku zwiększa się wykładniczo wraz ze wzrostem rozmiaru planszy. O skali tej trudności dobitnie świadczy fakt, że nadal nie jest znana ilość możliwych rozwiązań sudoku o rozmiarze planszy 16 na 16 pól, czyli następnego w kolejności po najpopularniejszej wersji planszy 9 na 9 pól.

## 1.2 Jaki algorytm?

Sudoku jest tego typu problemem, którego rozwiązywanie algorytmami naiwnymi jest wyjątkowo proste, ale jednocześnie praktycznie niemożliwe. "Algorytmami naiwnymi" nazywamy metody oparte na prostych, najczęściej intuicyjnych zasadach rozwiązywania problemu, będących łatwymi do zrozumienia i zaimplementowania. Przykładem algorytmu naiwnego dla porządkowania zbiorów jest sortowanie bąbelkowe o złożoności obliczeniowej  $O(n^2)$ .

Algorytm naiwny przy rozwiązywaniu sudoku oparty jest na metodzie brute force, polegającej na systematycznym sprawdzaniu wszystkich rozwiązań aż do znalezienia właściwego. Kolejne kroki algorytmu powodują przechodzenie przez wszystkie pola diagramu i wypełnianie ich kolejnymi liczbami. Dla sudoku o rozmiarze 9 na 9 oznacza to sprawdzenie 81 pól z 9 wariantami wartości w każdym z nich, co daje  $9^{81}$  możliwości w skrajnym przypadku, gdy rozwiązanie jest ostatnim ze sprawdzanych wariantów. Porównując do bardziej czytelnej reprezentacji można przyjąć, że  $9^{81}$  ma w przybliżeniu wartość  $2 \cdot 10^{77}$ . Jest to ogromna liczba, której skalę można opisać następującym przykładem: jeśli na sprawdzenie każdego z  $2 \cdot 10^{27}$  wariantów rozwiązań planszy zużyjemy 1 milisekundę, to sprawdzenie wszystkich zajmie w przybliżeniu  $10^{65}$  lat.

Niepraktyczność algorytmu naiwnego zachęca do użycia bardziej skomplikowanych metod. Krótki przegląd niektórych z nich znajduje się poniżej [A02,A03]:

- metoda backtrackingu - algorytm brute force można rozbudować o mechanizm cofania się (ang. backtracking), który pozwala na przerywanie rozwiązywania diagramów ze ślepych uliczkami, czyli takich, które po wprowadzeniu kolejnych liczb spowodowały naruszenie zasad poprawności rozwiązania,
- metoda eliminacji - algorytm ten opiera się na znanych strategiach rozwiązywania sudoku, np. na takich metodach jak Ukryty Singiel (ang. Hidden Single) czy Odkryty Singiel (ang. Naked Single). Oba mechanizmy opierają się na wyszukiwaniu takich pól diagramu, które mogą przyjąć tylko jedną możliwą wartość. Po wypełnieniu takiego pola stosuje się zasady sudoku do eliminacji możliwych wartości w pozostałych polach, po czym wraca się do wyszukiwania kolejnego pola z jedną możliwą wartością.
- metoda Dancing Links (DLX) Donalda Knutha - oparty jest na algorytmie X stosowanym dla rozwiązania problemu dokładnego pokrycia. Opiera się on na sprawdzonych mechanizmach przeszukiwania z nawrotami, ale dzięki nietypowej implementacji jest znacznie szybszy. Macierz przeszukiwania zbudowana jest na czterokierunkowej liście cyklicznej, której trójwymiarową reprezentację graficzną najlepiej oddaje powierzchnia torusa,
- metody mieszane - są to algorytmy oparte na połączeniu wyżej wymienionych metod.

Wszystkie wymienione dotychczas mechanizmy są deterministyczne, tzn. mają dokładnie określone kroki, a wynik ich działania jest określony jedynie przez warunki startowe. Oznacza to, że jeśli dana plansza sudoku ma rozwiązanie znajdujące się pod koniec wszystkich możliwych wariantów, to za każdym razem wymienione algorytmy będą działały z najdłuższym, skrajnie niekorzystnym dla nich czasem. Natomiast dla sudoku o dużych rozmiarach planszy rozwiązanie

może być niedostępne z powodu olbrzymiego narzutu czasowego koniecznego do znalezienia rozwiązania.

Próba obejścia tego problemu są algorytmy niedeterministyczne. Polegają one na wykorzystaniu losowości w sposobie wykonywaniu algorytmu lub jego operowaniu na danych. Nie gwarantują one uzyskania optymalnego wyniku, ale pozwalają na uzyskanie rozwiązania akceptowalnego w stosunkowo krótkim czasie. Krótki przegląd algorytmów niedeterministycznych znajduje się poniżej:

- metody stochastyczne - opierają się na wykorzystaniu losowości w procesie przeszukiwania przestrzeni rozwiązań. Przykładowe metody to: algorytm Monte Carlo, algorytm Random Walk.
- metody heurystyczne - opierają się na stosowaniu w algorytmach reguł opartych na zdrowym rozsądku, doświadczeniu, intuicji lub przybliżaniu ocen. Reguły te w zasadzie powinny doprowadzić do znalezienia jakiegoś rozwiązania, ale tego nie gwarantują. Przykładowe metody to: algorytm zachłanny, algorytm najbliższego sąsiada.
- metody metaheurystyczne - opierają się na metodach heurystycznych, ale kierowanych bardziej zaawansowanymi zbiorami reguł. Reguły te często są uogólnione i oderwane od konkretnego problemu, stanowiąc najczęściej zestaw wytycznych lub strategii do działania algorytmów heurystycznych. Przykładowe metody to: algorytmy genetyczne, algorytmy symulowanego wyżarzania, algorytmy rojów cząstek.

### 1.3 Dlaczego algorytmy genetyczne?

Algorytmy genetyczne to heurystyczne algorytmy przeszukiwania, które są inspirowane procesem ewolucji biologicznej. Są one często stosowane do rozwiązywania problemów optymalizacyjnych i przeszukiwania, gdzie tradycyjne metody są zbyt kosztowne obliczeniowo lub nieefektywne. Chociaż do rozwiązywania standardowego sudoku często używa się bardziej deterministycznych algorytmów, takich jak choćby rekurencyjny algorytm z cofaniem, to algorytmy genetyczne mogą być interesującym podejściem, gdy celem jest nie tyle rozwiązanie planszy, ale poznanie i zrozumienie zasad działania algorytmów genetycznych.

Sudoku jest wręcz stworzone do implementacji algorytmów genetycznych. Budowa planszy przekładająca się na prostą deklarację genów i chromosomów, proste zasady gry pozwalające na łatwe zdefiniowanie funkcji dopasowania, łatwość implementacji mechanizmów generowania pierwszego pokolenia oraz krzyżowania kolejnych bardzo ułatwiają wejście w świat algorytmów genetycznych. Dodatkowo istnienie wielu rozwiązań każdej planszy o nieco zmniejszonej ilości podanych pól stałych, przekładająca się na przestrzeń rozwiązań wypełnioną wieloma dostępnymi rozwiązaniami, powoduje zmniejszenie ryzyka utknięcia w lokalnym ekstremum. Intuicyjność

przełożenia zapisu chromosomów na sytuację na planszy sudoku pozwala na łatwą analizę rozwiązań pośrednich obserwowanych w trakcie generowania kolejnych pokoleń osobników.

## 2. Część teoretyczna

### 2.1 Język programowania Java

Java to język programowania, który zadebiutował w 1995 roku, stworzony przez firmę Sun Microsystems (obecnie Oracle). Jego początki sięgają projektu "Green" z 1991 roku, którego celem było stworzenie języka dla urządzeń elektronicznych. Ostatecznie Java została zaprojektowana z myślą o niezależności od platformy, co osiągnięto dzięki koncepcji Wirtualnej Maszyny Javy (JVM). Dzięki temu kod Java może być uruchamiany na dowolnym systemie operacyjnym, dla którego dostępna jest JVM, zgodnie z zasadą "napisz raz, uruchom wszędzie".

Do kluczowych zalet Javy należy jej przenośność, dojrzały ekosystem z bogatymi bibliotekami i frameworkami (np. Spring, Hibernate), silna orientacja obiektowa, automatyczne zarządzanie pamięcią (garbage collection) oraz wsparcie dla wielowątkowości. Java jest powszechnie wykorzystywana do tworzenia aplikacji korporacyjnych, aplikacji mobilnych (szczególnie na platformę Android), aplikacji webowych, systemów wbudowanych oraz w Big Data i analizie danych.

Cechami charakterystycznymi Javy są m.in. statyczne typowanie, co pomaga w wykrywaniu błędów na etapie kompilacji, silne wsparcie dla programowania obiektowego (enkapsulacja, dziedziczenie, polimorfizm), bezpieczeństwo (dzięki m.in. bytecode verification i sandbox environment) oraz wysoka wydajność (szczególnie w nowszych wersjach JVM). Pomimo swojej długiej historii, Java pozostaje jednym z najpopularniejszych i najczęściej używanych języków programowania na świecie. Jest także językiem ciągle rozwijanym i adaptowanym do nowych wyzwań technologicznych.

Java, pomimo swojej dojrzałości i rozbudowanych możliwości, charakteryzuje się stosunkowo niskim progiem wejścia dla początkujących programistów. Jej składnia, choć oparta na C++, jest bardziej uporządkowana i mniej podatna na typowe błędy. Dostępność obszernej dokumentacji, licznych tutoriali online oraz aktywna społeczność sprawiają, że nauka podstaw Javy jest przystępna. Koncepcje programowania obiektowego, które są fundamentem Javy, choć początkowo mogą wydawać się złożone, są dobrze wyjaśnione w wielu darmowych materiałach edukacyjnych.

Dodatkowym ułatwieniem dla początkujących jest dostępność darmowych i przyjaznych środowisk programistycznych (IDE). Przykłady popularnych, bezpłatnych IDE dla Javy to:

- **IntelliJ IDEA Community Edition:** potężne środowisko z wieloma funkcjami ułatwiającymi pisanie i debugowanie kodu, w darmowej wersji w pełni wystarczające do nauki,
- **Eclipse IDE:** kolejne popularne i darmowe środowisko, bardzo konfigurowalne i z dużą ilością wtyczek,

- **Visual Studio Code:** szeroko rozpowszechnione, darmowe środowisko programistyczne dla wielu języków programowania, wsparte bogatą kolekcją wtyczek,
- **NetBeans:** Darmowe IDE od Apache, proste w obsłudze i dobre dla początkujących.

Wszystkie wspomniane powyżej środowiska oferują między innymi kolorowanie składni, autouzupełnianie kodu, wbudowane debuggery, generatory diagramów, dostęp do repozytoriów i rozproszonych systemów kontroli wersji oraz wielu innych funkcjonalności.

## 2.2 Algorytmy genetyczne

### 2.2.1 Rys historyczny

Algorytmy genetyczne są stosunkowo starym podejściem do rozwiązywania problemów metodami algorytmicznymi. Ich prekursorem był John Holland z Uniwersytetu Michigan, który w latach 60-tych XX wieku rozpoczął badania nad systemami adaptacyjnymi. Jako jedną z inspiracji dla tych systemów obrał obserwowany szeroko w przyrodzie mechanizm adaptacji opartej na przekazywaniu informacji genetycznej kolejnym pokoleniom.

W 1975 roku wydał opublikował pracę pt. "Adaptation in Natural and Artificial Systems", która stanowiła fundament nowej teorii. Wprowadziła ona kluczowe koncepcje, takie jak reprezentacja chromosomowa, operatory krzyżowania i mutacji oraz selekcja oparta na funkcji przystosowania. Stanowiła ona bogate źródło wiedzy o algorytmach genetycznych, aktualne do dziś mimo pojawiających się coraz to nowych hipotez rozwijających tę dziedzinę, takich jak:

- hipoteza o schematach: opisująca mechanizm wykładniczego wzrostu ilości schematów w populacji pod wpływ działających z pokolenia na pokolenie mechanizmów selekcji, krzyżowania i mutacji. Schematami nazywamy odcinki chromosomów o krótkiej długości, niskim rzędzie i ponadprzeciętnym przystosowaniu.
- hipoteza o zbieżności: opisująca warunki dotyczące selekcji, operatorów genetycznych oraz reprezentacji, których spełnienie zwiększa zbieżność rozwiązań do globalnego optimum,
- hipoteza o dekonstrukcji i rekonstrukcji: wiążąca efektywność algorytmów genetycznych z cyklami rozkładania dobrych fragmentów chromosomu odpowiadających za dobre rozwiązania i ponownego ich składania w trakcie zmian pokoleń w ulepszone pakiety chromosomów wpływających na lepsze rozwiązania.
- hipotezy dotyczące wpływu parametrów: omawiają wpływ parametrów algorytmu genetycznego, takich jak wielkość populacji, czynniki krzyżowania czy prawdopodobieństwo mutacji, na jego wydajność.



W latach 90-tych nastąpiło szerokie rozpowszechnienie algorytmów genetycznych w różnych dziedzinach, napędzane między innymi pojawieniem się komercyjnych produktów informatycznych opartych na tej technologii. Do dnia dzisiejszego algorytmy genetyczne przeszły długą drogę, zaczynając od wczesnych zastosowań w problemach optymalizacji obejmujących między innymi projektowanie sieci rurociągów, sterowanie systemami i uczenie maszynowe, do obecnych szerokich zastosowań w takich dziedzinach jak inżynieria, ekonomia, biologia i sztuczna inteligencja.

### 2.2.2 Kluczowe pojęcia

Dla poprawnego i czytelnego opisywania metody algorytmów genetycznych konieczne jest wprowadzenie pewnych podstawowych pojęć, które przedstawiono poniżej [A05][A07]:

- **gen** to pojedynczy element w chromosomie reprezentujący pewną cechę rozwiązania. Właściwie dobrany gen powinien reprezentować jakąś składową lub parametr rozwiązania i posiadać określony zakres wartości. Ważnym kryterium jest także łatwość manipulacji na genie w operacjach krzyżowania i mutowania.
- **osobnik** (chromosom) to zbiór informacji o rozwiązaniu będący zbiorem pojedynczych genów. Każdy osobnik może być potencjalnym rozwiązaniem problemu - potencjalnym, gdyż z powodu specyfiki tworzenia opartej na metodach losowych, zapis rozwiązania może kolidować z zasadami opisującymi rozwiązywany problem.
- **populacja** to zbiór potencjalnych rozwiązań problemu reprezentowany przez pojedyncze osobniki. Wielkość populacji definiująca ilość osobników w pokoleniu jest jednym z parametrów wpływających na efektywność działania algorytmu.
- **funkcja przystosowania** to metoda oceny zgodności danego osobnika z warunkami będącymi rozwiązaniem problemu. Funkcja może być zrealizowana na kilka sposobów: albo w oparciu o zliczanie cech właściwych dla rozwiązania (wtedy najlepszy osobnik ma najwyższą w pokoleniu wartość tej funkcji), albo w oparciu o zliczanie naruszeń kolidujących z rozwiązaniem (wtedy najlepszy osobnik ma najniższą w pokoleniu wartość tej funkcji), albo mieszając powyższe podejścia. Właściwie dobrana funkcja charakteryzuje się korelacją z celem, dobrym różnicowaniem osobników, wysoką efektywnością obliczeniową oraz skalowalnością. Ponadto powinna funkcjonować w sposób unikający przedwczesnej zbieżności oraz zapewniający gładkość krajobrazu przystosowania (czyli unikanie lokalnych ekstremów, w głównej mierze zależnych od nadmiernego wpływu poszczególnych genów na wartości rozwiązania).
- **selekcja** to proces wybierania osobników z bieżącego pokolenia w celu użycia ich do utworzenia kolejnego pokolenia rozwiązań. Choć ogólną zasadą jest wybieranie osobników o

najlepszej wartości przystosowania, to w celu uniknięcia utknięcia w lokalnych ekstremach stosuje się także metody dopuszczające to puli rodzicielskiej osobniki o gorszym przystosowaniu. Dobra metoda selekcji powinna zapewniać równowagę pomiędzy dokładną eksploatacją już znalezionych rozwiązań a eksploracją nowych obszarów w przestrzeni rozwiązań. Aby to osiągnąć musi skutecznie balansować między presją selekcyjną, utrzymaniem różnorodności, stochastycznością i wydajnością obliczeniową.

- **krzyżowanie** to proces tworzenia nowego osobnika do kolejnego pokolenia na bazie dwóch wybranych osobników z bieżącego pokolenia. Krzyżowanie opiera się na założeniu, że z pewnym statystycznym prawdopodobieństwem istnieje możliwość przekazania osobnikowi potomnemu “dobrych” genów obu osobników rodzicielskich, co może potencjalnie doprowadzić do powstania chromosomu będącego bliżej rozwiązania. Istnieje wiele metod krzyżowania różniących się między sobą wpływem na presję selekcyjną, utrzymanie różnorodności czy stochastyczność.
- **mutacja** to proces losowej zmiany genów osobnika mający na celu zwiększenie różnorodności w pokoleniu, a tym samym zapobieganie utknięciu rozwiązań w lokalnym ekstremum. Dobra metoda mutująca powinna wprowadzać różnorodność osobników przy jednoczesnym unikaniu zbyt dużych zmian mogących “zepchnąć” potencjalnie dobre rozwiązanie zbyt daleko z aktualnej ścieżki ku optimum. Mutacja może być dynamicznie kontrolowana w trakcie generowania poszczególnych pokoleń.
- **pokolenie** (generacja) to zbiór osobników utworzonych w bieżącej iteracji algorytmu genetycznego. Pierwsze pokolenie jest generowane przy użyciu losowych lub heurystycznych, metod (rzadziej jest oparte na znanych rozwiązaniach cechujących się mniejszą optymalnością), a kolejne powstają przez krzyżowanie wybranych osobników z poprzedniego pokolenia.
- **warunki końcowe** definiują okoliczności zatrzymania działania algorytmu genetycznego. Idealnym warunkiem byłoby znalezienie optymalnego rozwiązania, ale stosowane są także warunki oparte na osiągnięciu pewnego poziomu wartości funkcji przystosowania. Ważnymi warunkami są warunki decydujące o zatrzymaniu procesu rozwiązywania problemu, oparte na osiągnięciu określonej ilości wygenerowanych pokoleń czy braku znaczących zmian w wartości funkcji dopasowania na przestrzeni kilku pokoleń.

### 2.2.3 Opis działania metody

Przed uruchomieniem algorytmu genetycznego konieczne są działania przygotowawcze obejmujące określenie struktury osobników i parametrów działania metody. Obejmują one między

innymi: zaprojektowanie chromosomu w sposób jak najlepiej odzwierciedlający problem, określenie funkcji oceniającej przystosowanie poszczególnych osobników oraz dobór metod selekcji i krzyżowania. Wstępnie wykonuje się też dobór parametrów takich jak wielkość populacji, prawdopodobieństwo mutacji oraz innych zależnych od przyjętych metod selekcji i krzyżowania osobników. Część z tych parametrów może być potem modyfikowana w trakcie działania algorytmu w celu jego dynamicznego dopasowania do uzyskiwanych wyników [A04][A05][A07].

Algorytm genetyczny opiera się na kilku powtarzalnych krokach, inspirowanych ewolucją biologiczną.

- krok 1 - wygenerowanie pierwszego pokolenia osobników,
- krok 2 - ocena jakości osobników przy użyciu funkcji przystosowania,
- krok 3 - sprawdzenie warunków końcowych na osobnikach bieżącego pokolenia:
  - jeśli zostały spełnione - zakończenie algorytmu,
  - jeśli nie zostały spełnione - przejście do kroku 3,
- krok 3 - użycie metod selekcji w celu wyboru osobników do puli rodzicielskiej,
- krok 4 - użycie metod krzyżowania w celu utworzenia nowego pokolenia osobników,
- krok 5 - narzucenie mutacji na nowe pokolenie osobników,
- krok 6 - powrót do kroku 2.

Elementy składowe algorytmów genetycznych, takie jak selekcja czy krzyżowanie, mogą realizować swoje cele w różny sposób. Poniżej opisano najpowszechniej używane metody:

- wybrane metody generowania pierwszego pokolenia:
  - **losowa** - opiera się na losowym doborze poszczególnych genów do chromosomu. Jest prosta w implementacji i zapewnia dużą różnorodność, ale przeważnie tworzy osobniki o niskim indeksie przystosowania,
  - **heurystyczna** - wymaga pewnej znajomości problemu i jego mniej optymalnych rozwiązań lub zaleceń wskazujących na drogę do rozwiązania. Utworzone na tej bazie metodami heurystycznymi osobniki pierwszego pokolenia mają zazwyczaj lepszy indeks funkcji przystosowania, ale często ograniczoną różnorodność.
  - **hybrydowa** - opiera się na łączeniu powyższych metod, co pozwala zachować sporą różnorodność przy zachowaniu wyższego średniego indeksu przystosowania.

- wybrane metody selekcji:

- **elitarna** - ta metoda jest najprostsza do zaimplementowania, a polega na wyborze grupy osobników o najwyższym indeksie dopasowania. Jej wadą jest podatność na dominację pojedynczych osobników mogąca prowadzić do utknięcia w lokalnym ekstremum.
- **rankingowa** - nieco podobna do elitarniej, gdyż także oparta na posortowanej wg funkcji przystosowania liście osobników. Różnicą jest przypisanie każdemu osobnikowi pewnej wartości wpływającej na prawdopodobieństwo wyboru, opartej na jego pozycji na liście. Jest mniej podatna od metody elitarniej na wpływ dominujących osobników, jednak może wolniej zbiegać ku potencjalnemu rozwiązaniu.
- **turniejowa** - opiera się na wielokrotnie powtarzanym turnieju, czyli losowym wyborze kilku osobników do listy pośredniej, z której osobnik o najlepszym przystosowaniu jest wybierany do puli rodziców. Jest łatwa w implementacji i pozwala unikać osobników słabym przystosowaniu zachowując jednocześnie presję selekcyjną. Może jednak prowadzić do utraty różnorodności osobników.
- **ruletki** - oparta jest przypisaniu każdemu osobnikowi współczynnika decydującego o prawdopodobieństwie wyboru opartego na wartości przystosowania (lepszy indeks przystosowania oznacza wyższą szansę wylosowania). Następnie losuje się pewną liczbę, która podczas przechodzenia przez listę osobników wskazuje osobnika wybranego. Jej zaletą i jednocześnie wadą jest promowanie osobników o najlepszym indeksie przystosowania, gdyż może prowadzić do zbyt szybkiego zbiegania w lokalnym ekstremum.

- wybrane metody krzyżowania:

- **metoda losowa** - oparta jest pobieraniu kolejnych genów od losowo wybranego rodzica. Charakteryzuje się wysokim potencjałem eksploracji i jest najłatwiejsza do zaimplementowania, jednak może niszczyć potencjalnie korzystne zależności między sąsiednimi genami,
- **metoda losowa z balansem** - również jest oparta na losowym doborze genów od obu rodziców, jednak kontroluje proces wyboru rodzica przez zwiększenie prawdopodobieństwa wyboru tego, który dotychczas przekazał mniej genów. Jej charakterystyka jest podobna do metody powyżej, przy czym potencjał eksploracji jest nieco mniejszy.

- metoda jednopunktowa - polega na podziale chromosomów rodziców na dwie części względem losowego punktu. Geny przed tym punktem są kopiowane od jednego rodzica, a geny za tym punktem od drugiego. Metoda ta charakteryzuje się prostotą implementacji oraz zachowaniem bloków genów sąsiadujących ze sobą. Może jednak być mniej efektywna w eksploracji przestrzeni rozwiązań, jeśli istotne dla rozwiązania cechy są rozproszone w chromosomie.
- metoda dwupunktowa - jest bardzo podobna do metody jednopunktowej, z tym, że chromosom dzieli się w dwóch losowych punktach. Początkowa i końcowa partia genów pobierana jest od jednego rodzica, a środkowa od drugiego. Charakterystyka metody jest podobna do metody jednopunktowej.

#### **2.2.4 Zalety i wady algorytmów genetycznych**

Algorytmy genetyczne posiadają wiele zalet [A08]:

- dzięki oparciu o losowość potrafią efektywnie eksplorować duże i złożone przestrzenie rozwiązań, w których tradycyjne metody optymalizacji mogą utknąć,
- dzięki mechanizmom takim jak mutacja i krzyżowanie są odporne na lokalne ekstrema, mają większą szansę na ucieczkę z lokalnych optimów i znalezienie globalnego rozwiązania,
- mogą być stosowane do szerokiej gamy problemów optymalizacyjnych w różnych dziedzinach dzięki swojej elastyczności i uniwersalności,
- umożliwiają równoległe przetwarzanie dzięki swojej naturze opartej na wielu populacjach, co może znacząco przyspieszyć proces optymalizacji,
- umożliwiają znajdowanie "wystarczająco dobrych" rozwiązań w rozsądnym czasie, co jest istotne w przypadkach, gdy znalezienie absolutnie optymalnego rozwiązania jest zbyt kosztowne obliczeniowo.

Niestety, algorytmy genetyczne mają także kilka wad [A08]:

- są metodami stochastycznymi, co oznacza, że nie ma pewności, iż znajdą najlepsze możliwe rozwiązanie,
- ich wydajność silnie zależy od odpowiedniego doboru parametrów, takich jak rozmiar populacji, prawdopodobieństwo krzyżowania i mutacji. Znalezienie optymalnych wartości tych parametrów może być trudne i czasochłonne,
- w przypadku złożonych problemów, algorytmy genetyczne mogą wymagać wielu iteracji i dużej populacji, co wiąże się z wysokim kosztem obliczeniowym,

- w przeciwieństwie do niektórych metod analitycznych, algorytmy genetyczne często dostarczają wynik, który jest trudny w interpretacji i ocenie, czy dane rozwiązanie jest wystarczająco dobre,
- nieodpowiedni dobór parametrów może doprowadzić algorytm do utknięcia w nieoptymalnym rozwiązaniu, zamknięcia się w ograniczonym zakresie przestrzeni rozwiązań lub braku różnorodności w populacji,
- sukces algorytmu genetycznego w dużym stopniu zależy od właściwie zdefiniowanej funkcji przystosowania, a zaprojektowanie takiej funkcji może być nietrywialne.

Podsumowując, algorytmy genetyczne są potężnym narzędziem do rozwiązywania problemów optymalizacyjnych, szczególnie tam, gdzie inne metody zawodzą. Jednak ich skuteczność zależy od odpowiedniego zastosowania i dostrojenia.

### **2.2.5 Obszary zastosowania**

Algorytmy genetyczne są wszechstronnym narzędziem optymalizacyjnym i mogą być stosowane w wielu różnych dziedzinach. Oto kilka przykładów ich zastosowania:

- inżynieria i projektowanie (np. optymalizacja kształtu i topologii, projektowanie układów elektronicznych, harmonogramowanie produkcji, projektowanie systemów sterowania, itp.),
- uczenie maszynowe i sztuczna inteligencja (np. wybór cech, optymalizacja hiperparametrów modeli uczenia maszynowego, ewolucja sieci neuronowych, itp.),
- logistyka i transport (np. znajdowanie najkrótszych tras, optymalizacja tras wielu pojazdów z uwzględnieniem ograniczeń, optymalizacja łańcucha dostaw, itp.),
- finanse (np. optymalizacja portfela inwestycyjnego, modelowanie finansowe i prognozowanie, itp.),
- biologia i bioinformatyka (np. składanie sekwencji DNA, projektowanie leków, modelowanie ewolucji, analiza ekspresji genów, itp.),
- robotyka (np. planowanie ścieżek robotów, uczenie zachowań robotów, itp.),
- gry i rozrywka (np. projektowanie strategii dla agentów AI w grach, itp.).

## 3. Część praktyczna

### 3.1 Specyfikacja projektu

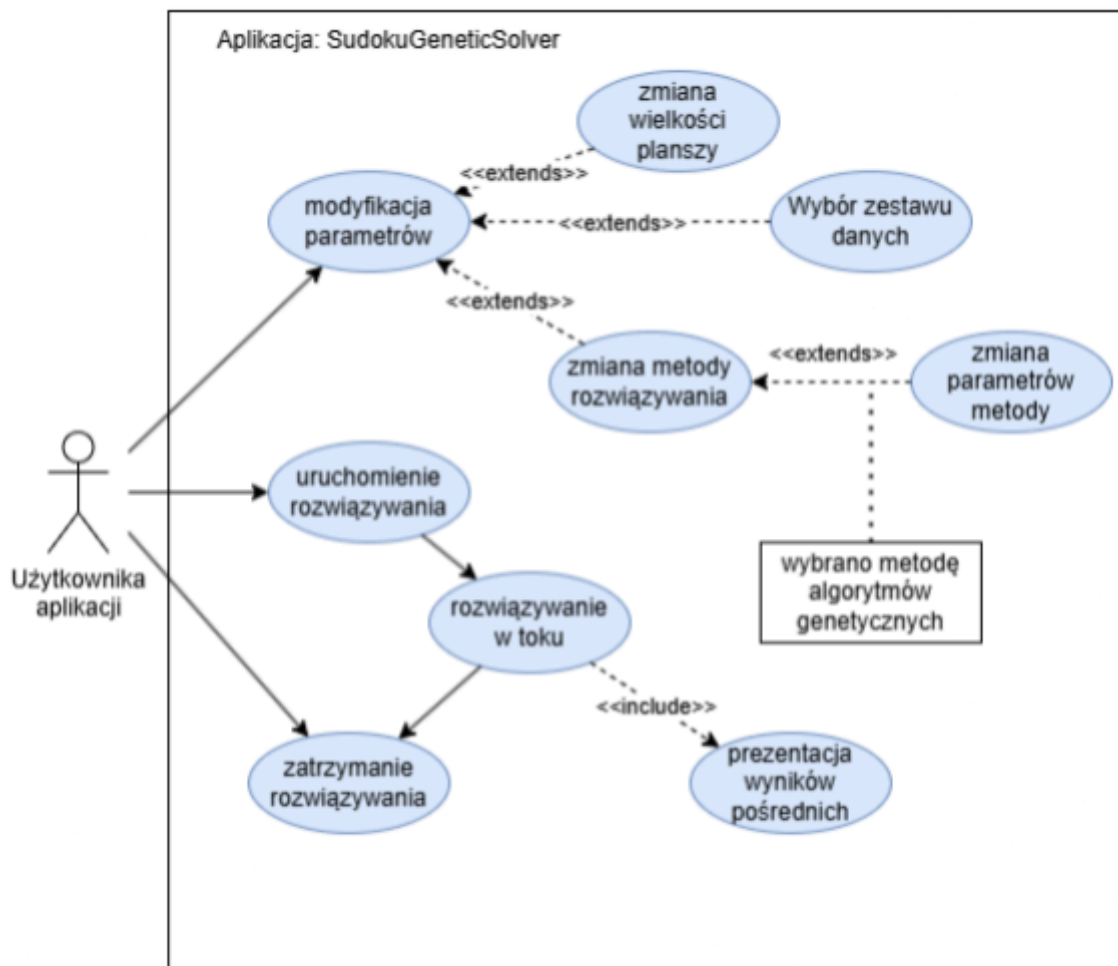
Na podstawie przeglądu literatury dotyczącej algorytmów genetycznych oraz istniejących w Internecie aplikacji do rozwiązywania sudoku zdefiniowano następujące wymagania tworzonej aplikacji:

- wymagania funkcjonalne:
  - możliwość wyboru parametrów (rozmiar planszy) i wariantów danych Sudoku,
  - możliwość wyboru metody rozwiązywania planszy: głównej (algorytmy genetyczne) oraz testowej (brute force),
  - możliwość sterowania parametrami metody algorytmów genetycznych,
  - możliwość zapisywania stanu procesu rozwiązania: skrócona (na ekranie) i rozszerzona (do pliku)
- wymagania niefunkcjonalne:
  - interfejs oparty na stylistyce aplikacji systemu MS DOS,
  - jednoekranowy interfejs zawierający planszę, menu oraz dane procesu rozwiązywania,
  - czytelna prezentacja planszy, ustawień aplikacji oraz danych z procesu rozwiązywania,
  - funkcjonalne menu pozwalające na szybki dostęp do opcji.

Poniżej przedstawiono diagram przypadku użycia aplikacji przez użytkownika w czasie wyboru ustawień i uruchomienia rozwiązywania sudoku. Użytkownik ma możliwość zmiany wielkości planszy sudoku, wyboru zestawu danych wejściowych oraz wyboru metody rozwiązywania. W przypadku wybrania metody algorytmów genetycznych dostępna staje się też możliwość wybrania parametrów działania tej metody. Po skonfigurowaniu, użytkownik uruchamia proces rozwiązywania, a aplikacja prezentuje wyniki pośrednie w trakcie jego trwania. Użytkownik może również w dowolnym momencie zatrzymać rozwiązywanie.

### 3.2 Specyfikacja elementów metody algorytmów genetycznych

Prace nad implementacją metody algorytmów genetycznych w aplikacji do rozwiązywania sudoku rozpoczęto od zdefiniowania kluczowego dla tej metody sposobu funkcjonowania genów.



Ilustracja 3.1: Diagram przypadków użycia

### 3.2.1 Geny

Przyjęto definicję genu jako pojedynczego pola planszy, mogącego przyjmować wartości zgodne z rozmiarem planszy. Przyjęcie takiej reprezentacji genu spełnia wszelkie zalecenia i dobre praktyki odnośnie doboru genów oraz gwarantuje łatwość implementacji metod ich krzyżowania i mutowania.

Zgodnie z powyższym założeniem pojedynczy osobnik będzie się składał z jednowymiarowej tablicy genów, których ilość będzie odpowiadała ilości pól planszy sudoku. Dla uproszczenia reprezentacji genów przyjęto, że pola stałe planszy, które nie powinny ulegać modyfikacjom w procesie krzyżowania czy mutowania, będą reprezentowane przez liczby ujemne odpowiadające wartości pola.

Przyjęta reprezentacja genu jest zgodna z zaleceniami prezentowanymi w literaturze, zapewniając łatwość manipulacji genetycznych, zwartość, adekwatność i jednoznaczną dekodowalność.



### 3.2.2 Metody generowania pierwszego pokolenia

W celu utworzenia pierwszego pokolenia osobników wybrano do implementacji dwie metody. W obu metodach losowanie wartości genów dotyczy tylko pól zmiennych, nie zawierających liczb ujemnych reprezentujących pola stałe.

Pierwsza z nich opiera się na całkowicie losowym generowaniu wartości genów z zakresu dozwolonych dla wielkości planszy liczb. Podczas losowania nie są sprawdzane reguły poprawności planszy sudoku, co oznacza między innymi, że w danym wierszu, kolumnie lub bloku mogą pojawić się wielokrotnie te same liczby. A jednocześnie mogą się trafić liczby, które nie wystąpią ani razu.

Druga metoda częściowo niweluje niedogodności pierwszej metody. Wprawdzie jest ona także oparta na losowym doborze wartości genów, ale w przeciwieństwie do pierwszej metody wylosowana wartość będzie sprawdzana pod względem naruszenia zasad sudoku w wierszu, kolumnie i bloku. W przypadku stwierdzenia kolizji losowanie będzie powtarzane określoną ilość razy.

Obie metody są zgodne z dostępnymi w literaturze zaleceniami, czyli gwarantują losowość, niskie obciążenie obliczeniowe, spore zróżnicowanie chromosomów poszczególnych osobników oraz spore pokrycie przestrzeni rozwiązań.

### 3.2.3 Metody obliczania funkcji dopasowania

Ocenę jakości osobnika w kontekście rozwiązania problemu realizuje funkcja obliczająca poziom jego przystosowania. Przyjęto dwie metody obliczania tej wartości, obie oparte na jednej z cech poprawnej planszy sudoku, według której każda liczba może wystąpić w danym wierszu, kolumnie lub bloku tylko raz.

Pierwsza metoda zlicza ilość brakujących wartości w każdym wierszu, kolumnie i bloku. Metoda spełnia literaturowe zalecenia odnośnie korelacji z celem optymalizacji oraz efektywności obliczeniowej, jednak ma problem ze mierzalnością, skalowaniem i różnicowaniem osobników (chromosomy z różną ilością powtarzających się liczb mogą mieć taką samą ocenę).

Druga metoda zlicza ilość kolizji generowanych przez wartość każdego genu w odpowiadających wierszach, kolumnach i blokach, przy czym kolizja z polem stałym ma wyższą wagę niż kolizja z wartością pola zmiennego. Takie podejście poprawia negatywne cechy metody pierwszej przy niewielkim obniżeniu efektywności obliczeniowej.

Obie metody opierają się na dążeniu do minimum, co oznacza, że im niższy wynik funkcji dopasowania, tym lepsza jakość osobnika. Idealny chromosom będący rozwiązaniem ma wartość funkcji przystosowania równą zero.

### 3.2.4 Metody selekcji rodziców

Metody selekcji są jednym z najważniejszych elementów w algorytmach genetycznych, gdyż w sporym stopniu wpływają one na proces rozwiązywania problemu. Z dostępnych w literaturze metod wybrano i zaimplementowano cztery: selekcję elitarną, ruletkową, rankingową i turniejową. Ich opis umieszczono w części teoretycznej niniejszej pracy.

### 3.2.5 Metody krzyżowania

W celu tworzenia nowych pokoleń na bazie poprzednich zaimplementowano cztery metody krzyżowania genów: dwie losowe i dwie oparte na bazie kopiowania całych bloków genów. Metody te noszą nazwy: losowa, losowa z balansem, jednopunktowa i dwupunktowa.

O ile trzy z nich są opisane w części teoretycznej, to metodę losowania z balansem trzeba tu przybliżyć. Opiera się ona na losowym doborze rodziców podczas kopiowania kolejnych genów, dodano w niej jednak mechanizm zliczający ilość wyborów każdego z rodziców. Na tej podstawie przy każdym kopiowaniu kolejnego genu zmienia się prawdopodobieństwo wyboru rodzica, co ma na celu zbalansowania proporcji źródeł pochodzenia genów.

Wszystkie metody są w różnym stopniu zgodne z zalecanymi w literaturze cechami, takimi jak eksploracja przestrzeni rozwiązań i eksploatacja lokalnego obszaru ekstremum. Jednak dzięki tej różnorodności pozwalają przetestować różne podejścia do tworzenia kolejnych pokoleń rozwiązań.

## 3.3 Realizacja aplikacji

Kod źródłowy aplikacji jest publicznie dostępny jako repozytorium w serwisie Github pod adresem <https://github.com/trawek1/SudokuGeneticSolver>.

### 3.3.1 Podział kodu źródłowego

Pod względem struktury kodu źródłowego aplikację podzielono na kilka części (do nazw klas dodano spacje dla zwiększenia czytelności):

- elementy implementujące planszę sudoku - składającą się z klas: Board Base, Board Field, Board Field Possibilities i Board,
- elementy implementujące metody rozwiązujące - składająca się z klas: Solver Base, Solver Brute Force, Solver Genetic Parent Maker, Solver Genetic Individual, Solver Genetic Crossover i Solver Genetic Population,

- elementy pomocnicze - składające się z klas typu enum: Sudoku Sizes, Solving Methods, Parent Generating Methods, Parent Selection Methods, Crossover Methods, Fitness Calculating Methods.
- elementy spinające całość: App, Sudoku Solver GUI.
- elementów realizujących testowanie: pliki o nazwach utworzonych wg wzorca <NazwaKlasy>Test.



Ilustracja 3.2: Diagram klas aplikacji Sudoku Genetic Solver

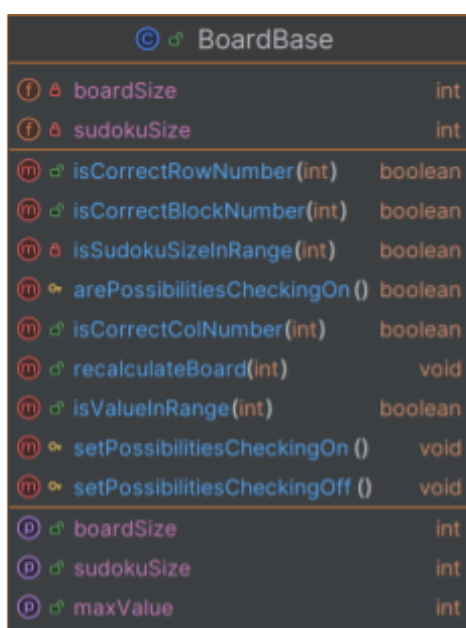
- biblioteki zewnętrzne:
  - Lanterna v3.1.0 - to biblioteka Java oferująca interfejs tekstowy CLI, obsługująca terminale XTerm oraz ich emulatory. Może działać w trybie terminala niskiego poziomu, oferującego jedynie proste metody zmiany poszczególnych pól, lub w trybie GUI oferującym zestaw komponentów typu przyciski, okna, etykiety, itp.
  - Tinylog v2.7.0 - to lekka biblioteka logowania dla języka Java. Charakteryzuje się prostotą użycia i konfiguracji oraz wysoką wydajnością. Oferuje równoległe logowanie do kilku miejsc docelowych (konsola, plik), filtrowanie komunikatów na każdym kanale osobno, szerokie możliwości dopasowania i konfiguracji. Wspiera podstawowe poziomy ważności komunikatów (track, debug, info, warn, error).
  - JUnit v5 - to popularna biblioteka obsługująca przeprowadzanie testów jednostkowych i integracyjnych kodu w Javie. Oferuje nowoczesne i elastyczne środowisko testowe, pozwalające na automatyczne tworzenie testów i ich wykonywanie. Wersja 5 wprowadza nowe funkcjonalności, między innymi testy parametryzowane czy tagi.

### 3.3.2 Przegląd głównych klas aplikacji

Podstawową grupą klas dla aplikacji są klasy o nazwach zaczynających się słowem Board. Zawierają one logikę związaną z obsługą planszy sudoku, sprawdzaniem poprawności wypełnienia, wyszukiwaniem kolizji i eksportem danych do klas solvera.

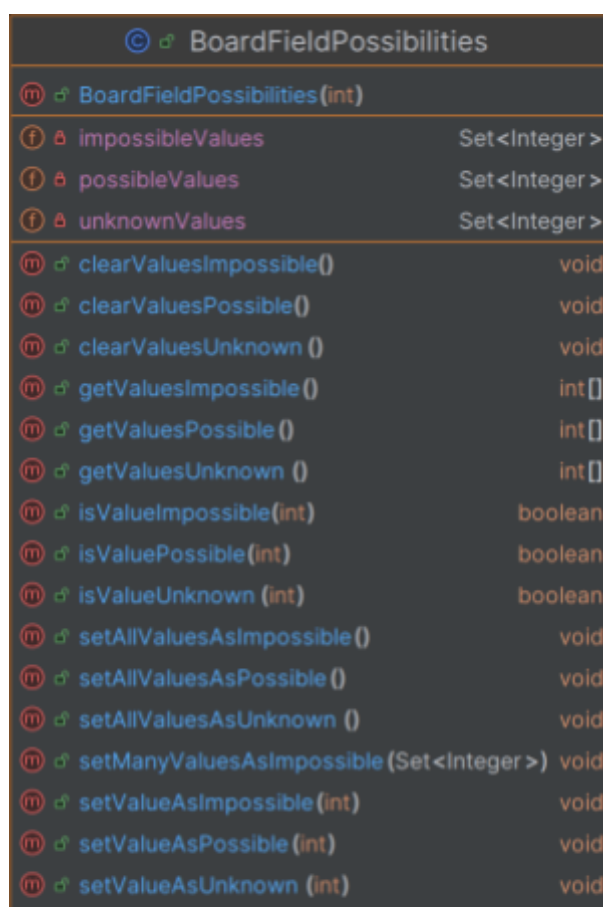
#### BoardBase

Klasę zaprojektowano jako klasę bazową, która będzie dziedziczona przez pozostałe klasy związane z funkcjonalnością planszy sudoku. Klasa zawiera metody wspólne dla wszystkich klas Board\*, realizujące między innymi kontrolę poprawności podanych parametrów (numer wiersza czy kolumny, wartość pola, rozmiar planszy) czy włączające lub blokujące mechanizm kontroli możliwych wartości pól.



BoardBase	
boardSize	int
sudokuSize	int
isCorrectRowNumber(int)	boolean
isCorrectBlockNumber(int)	boolean
isSudokuSizeInRange(int)	boolean
arePossibilitiesCheckingOn()	boolean
isCorrectColNumber(int)	boolean
recalculateBoard(int)	void
isValueInRange(int)	boolean
setPossibilitiesCheckingOn()	void
setPossibilitiesCheckingOff()	void
boardSize	int
sudokuSize	int
maxValue	int

Ilustracja 3.3: Właściwości i metody klasy BoardBase



BoardFieldPossibilities	
BoardFieldPossibilities(int)	
impossibleValues	Set<Integer>
possibleValues	Set<Integer>
unknownValues	Set<Integer>
clearValuesImpossible()	void
clearValuesPossible()	void
clearValuesUnknown()	void
getValuesImpossible()	int[]
getValuesPossible()	int[]
getValuesUnknown()	int[]
isValueImpossible(int)	boolean
isValuePossible(int)	boolean
isValueUnknown(int)	boolean
setAllValuesAsImpossible()	void
setAllValuesAsPossible()	void
setAllValuesAsUnknown()	void
setManyValuesAsImpossible(Set<Integer>)	void
setValueAsImpossible(int)	void
setValueAsPossible(int)	void
setValueAsUnknown(int)	void

Ilustracja 3.4: Właściwości i metody klasy BoardFieldPossibilities

#### BoardFieldPossibilities

Klasa dostarcza funkcjonalność związaną z kontrolą potencjalnych wartości pola pod kątem ich kolizji z wartościami w wierszu, kolumnie i bloku. Potencjalne wartości są przypisane do jednego z trzech statusów:

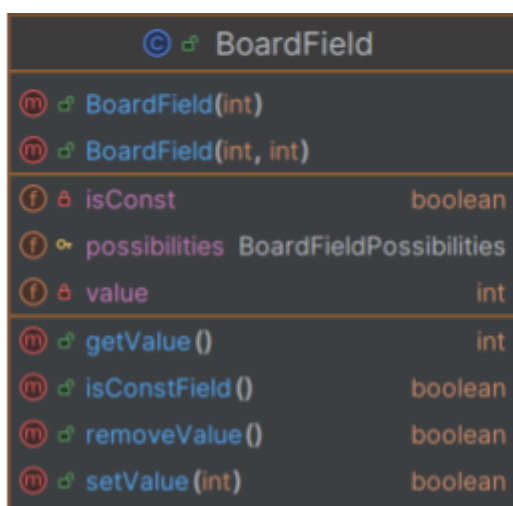
- status niemożliwy - dotyczy wartości już występujących w polach wiersza, kolumny lub bloku,

- status możliwy - obejmuje wartości, o których wiadomo, że na pewno nie występują w polach wiersza, kolumny i bloku,
- status nieznany - dotyczy wartości, których występowanie w polach wiersza, kolumny lub bloku nie zostało stwierdzone, ale nie może być wykluczone.

## BoardField

Klasa dostarcza funkcjonalności związane z pojedynczym polem planszy sudoku. Przechowuje wartość pola, opcje możliwych wartości oraz informacje o zmienności pola oraz dostarcza metody pozwalające manipulować tymi cechami.

Kontrolę wartości potencjalnych dla wszystkich pól planszy można włączyć lub wyłączyć metodami dostępnymi w klasie bazowej `BoardBase`, po której niniejsza klasa dziedziczy.



BoardField	
BoardField(int)	
BoardField(int, int)	
isConst	boolean
possibilities	BoardFieldPossibilities
value	int
getValue()	int
isConstField()	boolean
removeValue()	boolean
setValue(int)	boolean

Ilustracja 3.5: Właściwości i metody klasy `BoardField`



Board	
Board(int[])	
Board(int)	
board	BoardField[][]
getBoardData()	int[]
getUsedValuesFromBlock(int, int)	Set<Integer>
getUsedValuesFromCol(int)	Set<Integer>
getUsedValuesFromRow(int)	Set<Integer>
getValue(int, int)	int
isConstField(int, int)	boolean
isEmptyField(int, int)	boolean
removeValue(int, int)	boolean
setBoardData(int[])	void
setValue(int, int, int)	boolean
xxx_showBoard()	void

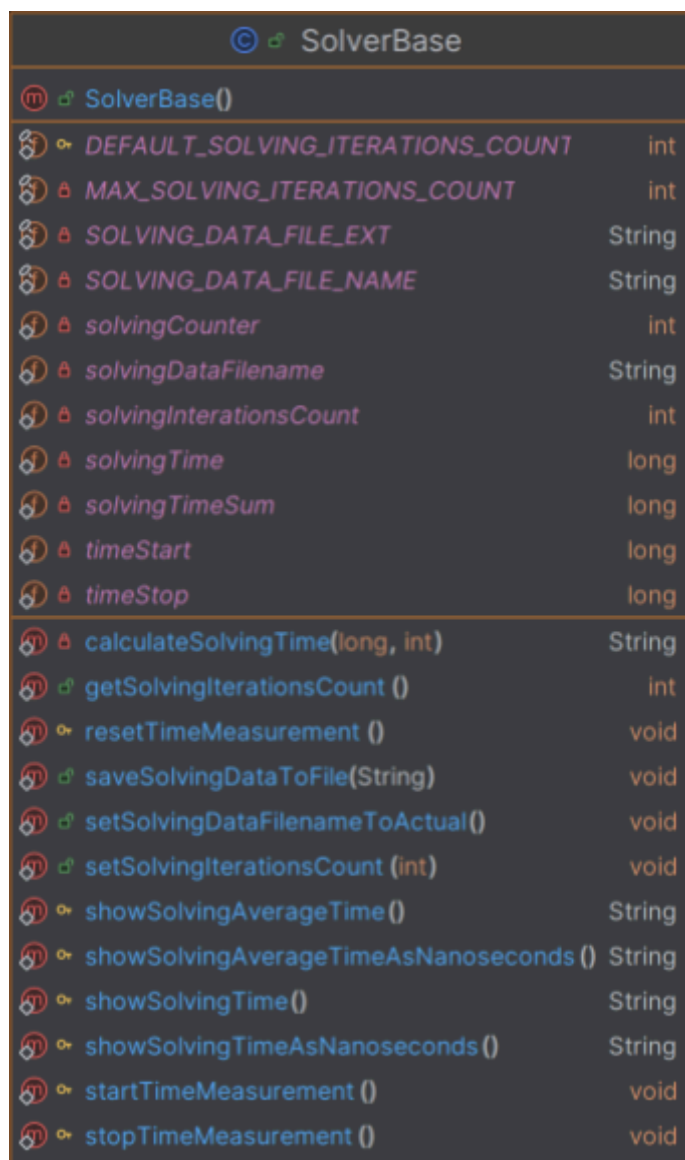
Ilustracja 3.6: Właściwości i metody klasy `Board`

## Board

Klasa skupiająca wszystkie pozostałe klasy obsługujące planszę sudoku. W dwuwymiarowej tablicy przechowuje pola planszy, a konstruktory umożliwiają utworzenie obiektu przez podanie rozmiaru planszy lub tablicy z przygotowanymi danymi rozmieszczenia wartości na planszy. Poza metodami odziedziczonymi i zarządzającymi danymi na planszy udostępnia też prostą metodę prezentującą w konsoli wygląd planszy, dzięki czemu ułatwiony jest proces dokonywania i oceny zmian w kodzie klasy.

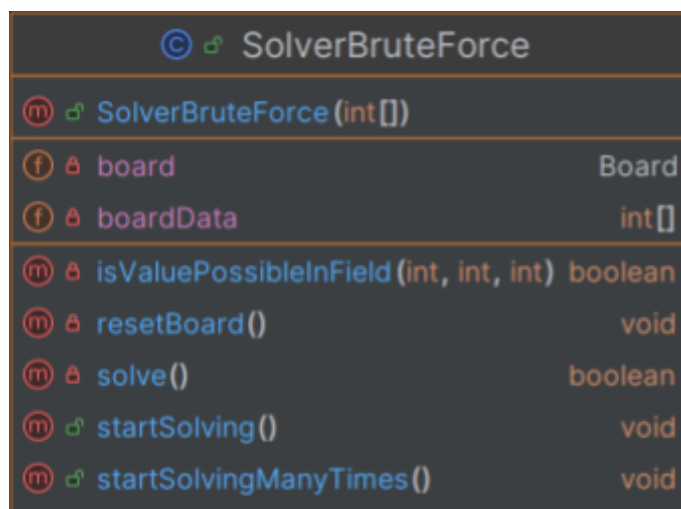
## SolverBase

Klasa SolverBase służy jako bazowa klasa dla różnych algorytmów rozwiązujących Sudoku umieszczonych w plikach, których nazwa rozpoczyna się od wyrazu Solver. Zawiera funkcjonalności wspólne dla tych algorytmów, takie jak pomiar czasu rozwiązywania i zapisywanie parametrów pracy do pliku. Definiuje stałe związane z plikiem danych pośrednich algorytmu genetycznego oraz domyślną i maksymalną liczbą iteracji rozwiązywania planszy w celu uśrednienia pomiarów czasu.



© SolverBase	
Ⓜ SolverBase()	
Ⓜ DEFAULT_SOLVING_ITERATIONS_COUNT	int
Ⓜ MAX_SOLVING_ITERATIONS_COUNT	int
Ⓜ SOLVING_DATA_FILE_EXT	String
Ⓜ SOLVING_DATA_FILE_NAME	String
Ⓜ solvingCounter	int
Ⓜ solvingDataFilename	String
Ⓜ solvingIterationsCount	int
Ⓜ solvingTime	long
Ⓜ solvingTimeSum	long
Ⓜ timeStart	long
Ⓜ timeStop	long
Ⓜ calculateSolvingTime(long, int)	String
Ⓜ getSolvingIterationsCount()	int
Ⓜ resetTimeMeasurement()	void
Ⓜ saveSolvingDataToFile(String)	void
Ⓜ setSolvingDataFilenameToActual()	void
Ⓜ setSolvingIterationsCount(int)	void
Ⓜ showSolvingAverageTime()	String
Ⓜ showSolvingAverageTimeAsNanoseconds()	String
Ⓜ showSolvingTime()	String
Ⓜ showSolvingTimeAsNanoseconds()	String
Ⓜ startTimeMeasurement()	void
Ⓜ stopTimeMeasurement()	void

Ilustracja 3.7: Właściwości i metody klasy SolverBase



© SolverBruteForce	
Ⓜ SolverBruteForce(int[])	
Ⓜ board	Board
Ⓜ boardData	int[]
Ⓜ isValuePossibleInField(int, int, int)	boolean
Ⓜ resetBoard()	void
Ⓜ solve()	boolean
Ⓜ startSolving()	void
Ⓜ startSolvingManyTimes()	void

Ilustracja 3.8: Właściwości i metody klasy SolverBruteForce

## SolverBruteForce

Klasa realizuje rozwiązanie planszy sudoku metodą rekurencyjną przez kolejne wypełnianie wolnych pól dostępnymi liczbami. W przypadku uzyskania kolizji i wyczerpania zasobu dostępnych dla pola liczb wraca do poprzednich wywołań i modyfikuje tam ustawione wartości. Klasa

reprezentuje algorytm naiwny o wykładniczej złożoności obliczeniowej, co czyni ją bardzo podatną na zmiany wielkości planszy.

## SolverGeneticParentMaker

Klasa realizuje pierwszy etap działania algorytmu genetycznego polegający na wygenerowaniu pierwszego pokolenia. W tym celu implementuje dwie metody, z których jedna generuje dane w sposób całkowicie losowy, a druga używa wykrywania kolizji i sprawdzaniem poprawności wylosowanych liczb.

SolverGeneticParentMaker	
SolverGeneticParentMaker (int[])	
MAX_SAFE_GENERATION_ATTEMPTS	int
PARENT_GENERATING_METHOD_DEFAULT	ParentGeneratingMethods
board	Board
parentGeneratingMethod	ParentGeneratingMethods
random	Random
changeParentGeneratingMethod ()	void
generateParent ()	void
generatorRandomFull ()	void
generatorRandomWithChecking ()	void
getBoardData()	int[]
getParentGeneratingMethodName ()	String
saveSolvingPreferencesToFile ()	void
xxx_showBoard()	void

Ilustracja 3.9: Właściwości i metody klasy SolverGeneticParentMaker

SolverGeneticIndividual	
SolverGeneticIndividual (int[])	
FITNESS_CALCULATING_METHOD_DEFAULT	FitnessCalculatingMethods
FITNESS_PENALTY_CONST_COLLISION	int
FITNESS_PENALTY_EMPTY_FIELD	int
FITNESS_PENALTY_VALUE_COLLISION	int
board	Board
boardErrorLevel	int
fitnessCalculatingMethod	FitnessCalculatingMethods
initialBoardData	int[]
calculateFitness ()	void
calculateFitnessByMissingValues ()	void
calculateFitnessByValuesCollisions ()	void
changeFitnessCalculatingMethod ()	FitnessCalculatingMethods
getBoardData()	int[]
getBoardErrorLevel()	int
getBoardFromParent()	void
getFitnessCalculatingMethodName ()	String
getInitialBoardData()	int[]
saveSolvingPreferencesToFile ()	void
xxx_showBoard()	void

Ilustracja 3.10: Właściwości i metody klasy SolverGeneticIndividual

## SolverGeneticIndividual

Klasa przechowuje dane pojedynczego osobnika oraz implementuje dwie metody funkcji przystosowania. Pierwsza z nich ocenia osobnika na podstawie ilości brakujących na planszy wartości, natomiast druga na podstawie ilości kolizji wygenerowanych przez wartości w polach planszy.

## SolverGeneticCrossover

Klasa implementuje cztery metody krzyżowania osobników rodzicielskich w celu utworzenia osobników potomnych dla kolejnego pokolenia. Dwie metody oparte są na podejściu losowym, przy czym jedna z nich wprowadza współczynnik balansujący kopiowanie genów od poszczególnych rodziców. Dwie kolejne metody bazują na podziale chromosomów rodziców odpowiednio w jednym



lub dwóch punktach. Ponadto klasa implementuje metodę odpowiadającą za wystąpienie mutacji na skopiowanym chromosomie pojedynczego osobnika.

© SolverGeneticCrossover	
☞ SolverGeneticCrossover (int[], int[])	
☞ CROSSOVER_METHOD_DEFAULT CrossoverMethods	
☞ MUTATION_PROBABILITY_DEFAULT	int
☞ MUTATION_PROBABILITY_MAX	int
☞ MUTATION_PROBABILITY_MIN	int
☞ MUTATION_PROBABILITY_STEF	int
☞ RANDOM_BOOLEAN_BALANCE_MAX	int
☞ RANDOM_BOOLEAN_BALANCE_MIN	int
☞ USE_RANDOM_POINT	int
☞ USE_SYMMETRIC_POINT	int
☞ boardLength	int
☞ childBoardData	int[]
☞ crossoverMethod	CrossoverMethods
☞ fatherBoardData	int[]
☞ motherBoardData	int[]
☞ mutationProbability	int
☞ randomBooleanBalance	int
☞ changeCrossoverMethod ()	void
☞ changeMutationProbabilityByStep ()	void
☞ crossoverSinglePoint (int)	int[]
☞ crossoverTwoPoint (int, int)	int[]
☞ crossoverUniform ()	int[]
☞ crossoverUniformBalanced ()	int[]
☞ getCrossover ()	int[]
☞ getCrossoverMethodName ()	String
☞ getMutationProbability ()	int
☞ getRandomBalancedBoolean ()	boolean
☞ getRandomPoint ()	int
☞ getSymmetricSinglePoint ()	int
☞ getSymmetricTwoPointLeft ()	int
☞ getSymmetricTwoPointRight ()	int
☞ isCrossoverPointNotInRange (int)	boolean
☞ makeMutations (int[])	int[]
☞ saveSolvingPreferencesToFile ()	void
☞ setCrossoverMethod (CrossoverMethods)	void
☞ setMutationProbability (int)	void

Ilustracja 3.11: Właściwości i metody klasy SolverGeneticCrossover

© SolverGeneticPopulation	
☞ SolverGeneticPopulation (int[])	
☞ BEST_PARENTS_PERCENT_DEFAULT	int
☞ BEST_PARENTS_PERCENT_MAX	int
☞ BEST_PARENTS_PERCENT_MIN	int
☞ BEST_PARENTS_PERCENT_STEF	int
☞ PARENT_SELECTION_METHOD_DEFAULT ParentSelectionMethods	
☞ POPULATION_SIZE_DEFAULT	int
☞ POPULATION_SIZE_MAX	int
☞ POPULATION_SIZE_MIN	int
☞ POPULATION_SIZE_STEP	int
☞ bestParentsPercent	int
☞ generationsCount	int
☞ initialBoard	int[]
☞ parentSelectionMethod	ParentSelectionMethods
☞ population	List<SolverGeneticIndividual>
☞ populationSize	int
☞ changeBestParentsPercentByStep ()	void
☞ changeParentSelectionMethod ()	void
☞ changePopulationSizeByStep ()	void
☞ createNextAndSwapPopulation ()	void
☞ getBestParentsPercent ()	int
☞ getGenerationsCount ()	int
☞ getNumberOfParents ()	int
☞ getParentSelectionMethodName ()	String
☞ getParentsByEliteSelection ()	List<SolverGeneticIndividual>
☞ getParentsByRouletteSelection ()	List<SolverGeneticIndividual>
☞ getPopulationSize ()	int
☞ getStatsAverageFitness ()	double
☞ getStatsBestFitness ()	int
☞ getStatsFitnessOfLastParent ()	int
☞ getStatsWorstFitness ()	int
☞ initializePopulation ()	void
☞ saveSolvingPreferencesToFile ()	void
☞ setBestParentsPercent (int)	void
☞ setPopulationSize (int)	void
☞ sortPopulationByFitness ()	void

Ilustracja 3.12: Właściwości i metody klasy SolverGeneticPopulation

## SolverGeneticPopulation

Klasa służy do zarządzania kolejnymi populacjami chromosomów generowanych w trakcie rozwiązywania sudoku metodą algorytmów genetycznych. Udostępnia właściwości związane z populacją, takie jak liczebność populacji, procent rodziców pobieranych z poprzedniego pokolenia



czy licznik generacji. Udostępnia również metody pobierające dane statystyczne o dopasowaniu osobników z populacji czy rozpoczynające proces selekcji rodziców lub krzyżowania.

**SudokuSizes, SolvingMethods, ParentGeneratingMethods,**

**ParentSelectionMethods, FitnessCalculatingMethods, CrossoverMethods**

Wszystkie wymienione powyżej klasy są klasami typu ENUM i mają za zadanie ułatwić operowanie za pomocą menu ustawieniami poszczególnych parametrów. W tym celu we wszystkich zaimplementowano dwie metody: jedna z nich pozwala wyświetlać czytelną nazwę wybranej stałej, a druga umożliwia rotacyjną zmianę stałych.

SudokuSizes	
<b>SudokuSizes(String)</b>	
X2	
X3	
X4	
X5	
displayName	String
getDisplayName()	String
next()	SudokuSizes
valueOf(String)	SudokuSizes
values()	SudokuSizes[]

Ilustracja 3.13: Właściwości i metody klasy enum SudokuSizes

ParentSelectionMethods	
<b>ParentSelectionMethods(String)</b>	
ELITE_SELECTION	
ROULETTE_SELECTION	
displayName	String
getDisplayName()	String
next()	ParentSelectionMethods
valueOf(String)	ParentSelectionMethods
values()	ParentSelectionMethods[]

Ilustracja 3.15: Właściwości i metody klasy enum ParentSelectingMethods

SolvingMethods	
<b>SolvingMethods(String)</b>	
BRUTE_FORCE	
GENETIC	
displayName	String
getDisplayName()	String
next()	SolvingMethods
valueOf(String)	SolvingMethods
values()	SolvingMethods[]

Ilustracja 3.14: Właściwości i metody klasy enum SolvingMethods

CrossoverMethods	
<b>CrossoverMethods(String)</b>	
CROSSOVER_BALANCED_UNIFORM	
CROSSOVER_SINGLE_POINT	
CROSSOVER_TWO_POINTS	
CROSSOVER_UNIFORM	
displayName	String
getDisplayName()	String
next()	CrossoverMethods
valueOf(String)	CrossoverMethods
values()	CrossoverMethods[]

Ilustracja 3.16: Właściwości i metody klasy enum CrossoverMethods

ParentGeneratingMethods	
ParentGeneratingMethods (String)	
RANDOM_FULL	
RANDOM_WITH_CHECKING	
displayName	String
getDisplayName()	String
next()	ParentGeneratingMethods
valueOf(String)	ParentGeneratingMethods
values()	ParentGeneratingMethods []

Ilustracja 3.17: Właściwości i metody klasy enum ParentGeneratingMethods

FitnessCalculatingMethods	
FitnessCalculatingMethods (String)	
MISSING_VALUES	
VALUES_COLLISIONS	
displayName	String
getDisplayName()	String
next()	FitnessCalculatingMethods
valueOf(String)	FitnessCalculatingMethods
values()	FitnessCalculatingMethods []

Ilustracja 3.18: Właściwości i metody klasy enum FitnessCalculatingMethods

## App, SudokuSolverGUI

Obie klasy służą do uruchamiania projektu. Klasa App opiera się na użyciu konsoli w celu monitorowania procesu działania aplikacji i jako taka jest przeznaczona bardziej do użytkowania w czasie pisania i testowania kodu. Klasa SudokuSolverGUI uruchamia środowisko graficzne biblioteki CLI Lanterna i zarządza nim podczas działania aplikacji. W tym celu implementuje metody rysujące nagłówek, planszę, menu i tabelę wyników pośrednich oraz wywołuje metody zawierające logikę aplikacji.

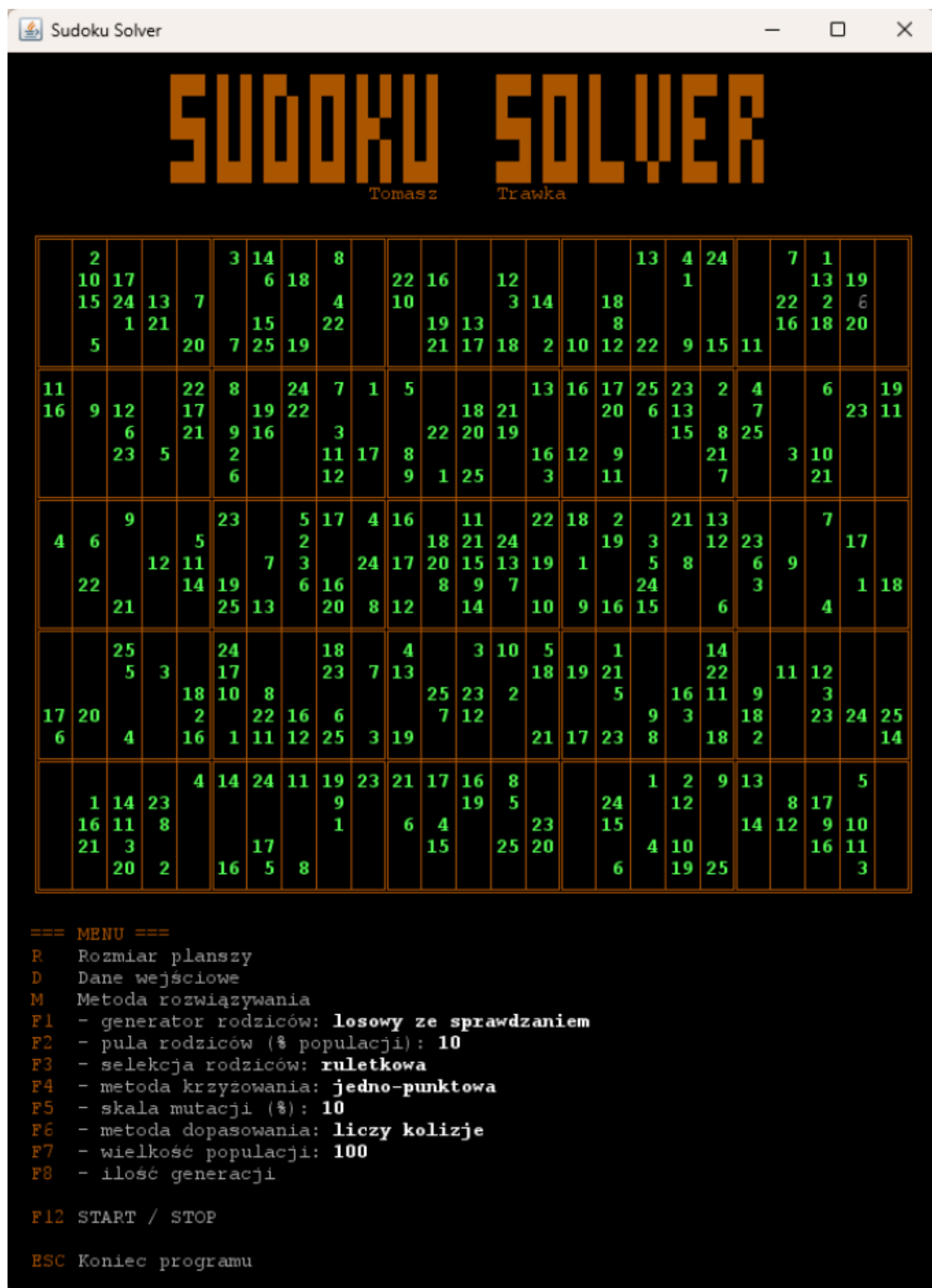
### Klasy testowe \*Test

Przegląd klas aplikacji kończą klasy służące do testowania aplikacji. Są oparte na powszechnie używanej bibliotece JUnit i z racji standardowej konstrukcji nie będą tu przedstawiane. Ich celem jest przetestowanie każdej funkcjonalności aplikacji pod kątem zarówno prawidłowych jak i nieprawidłowych zachowań aplikacji i użytkownika.

### 3.3.3 Interfejs graficzny aplikacji

Zgodnie z założeniami projektowymi, interfejs graficzny aplikacji zaimplementowano w stylu aplikacji jednoekranowej. Pozwoliło to zachować jego przejrzystość oraz intuicyjność sterowania. Ekran został podzielony na cztery części:

- obszar stałego nagłówka z nazwą aplikacji,
- obszar adaptowalnej planszy, która w zależności od rozmiaru ustawia się centralnie względem szerokości okna,
- obszar menu
- obszar prezentacji danych pośrednich pojawiających się w trakcie rozwiązywania planszy.



Ilustracja 3.19: Aplikacja Sudoku Solver uruchomiona w trybie graficznym

## 4. Podsumowanie projektu

Rezultatem powyższej pracy jest aplikacja Sudoku Solver umożliwiającą rozwiązywanie plansz sudoku metodą algorytmów genetycznych oraz metodą rekurencyjną podstawiania z cofaniem. W fazie projektowej określono wymagania odnośnie aplikacji, jej funkcjonalności i obszar działania. Określono także interfejs aplikacji, sposób komunikacji z użytkownikiem oraz sposób prezentacji wyników. W fazie realizacji wykonano:

- po stronie back-endu:
  - implementacji klas i funkcjonalności odpowiedzialnych za logikę planszy,
  - implementacji klas i funkcjonalności odpowiedzialnych za logikę algorytmu genetycznego,
  - implementacji klasy i funkcjonalności odpowiedzialnych za logikę algorytmu brute force,
  - implementacji klasy i funkcjonalności odpowiedzialnych za logikę funkcjonalności pomocniczych, jak pomiar czasu czy zapis danych z kolejnych pokoleń do pliku csv,
  - dołączenie biblioteki JUnit i implementację klas testowych,
  - dołączenie biblioteki tinylog i implementację systemu komunikatów,
- po stronie front-endu:
  - włączenie i przetestowanie biblioteki Command Line Interface Lanterna,
  - implementacja jednoekranowego interfejsu komunikacyjnego prezentującego menu, planszę oraz wyniki pracy programu.

Po ukończeniu programu przeprowadzono testy dla kolejnych rozmiarów plansz sudoku, których wyniki (uśrednione czasy rozwiązania planszy) przedstawiono w tabeli poniżej.

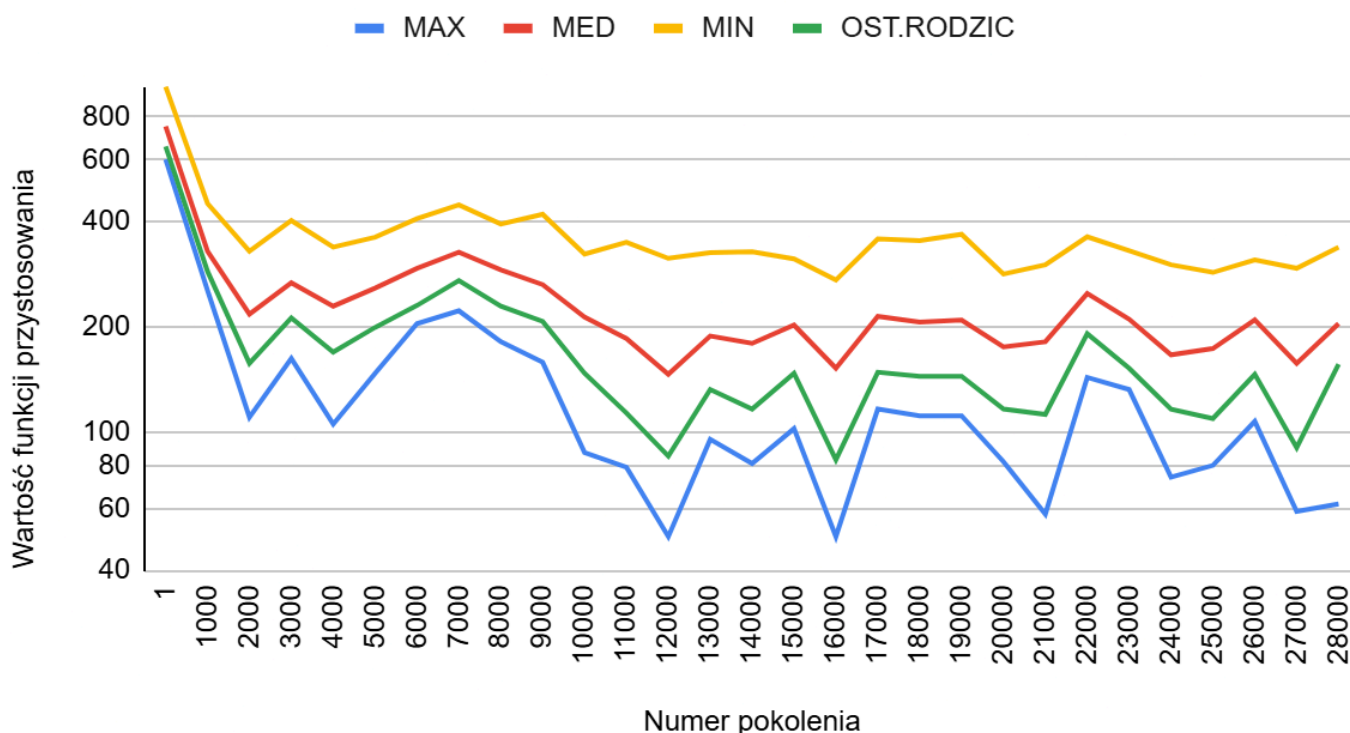
Rozmiar planszy	Metoda: brute force	Metoda: algorytmy genetyczne
sudoku 2x2 (4x4, 16 pól)	00m 00s 000ms 069566ns	do dodania
sudoku 3x3 (9x9, 81 pól)	00m 00s 118ms 544300ns	do dodania
sudoku 4x4 (16x16, 256 pól)	02m 40s 617ms 519533ns	do dodania
sudoku 5x5 (25x25, 625 pól)	zatrzymano po 20 minutach	do dodania

W celu zaprezentowania sposobu działania algorytmu genetycznego zebrano dane z plików csv gromadzących informacje o kolejnych pokoleniach i zestawiono w postaci poniższego wykresu. Rozwiązanie realizowano dla następujących parametrów:

- rozmiar planszy sudoku: 4x4

- metoda generowania rodziców: losowa ze sprawdzaniem,
- metoda obliczania dopasowania: ilość kolizji wartości,
- metoda krzyżowania: losowa z balansem,
- metoda selekcji rodziców: ruletkowa,
- wielkość populacji: 100 osobników
- wielkość puli rodziców: 10% / 10 osobników

## Wartości funkcji przystosowania w kolejnych pokoleniach



Mimo niekompletnego przetestowania wszystkich wariantów zadania można, na podstawie powyższych wyników częściowych oraz powyższej pracy, wyciągnąć następujące wnioski:

- dla małych rozmiarów plansz metody deterministyczne są zdecydowanie bardziej wydajne od algorytmów genetycznych,
- dobór właściwych parametrów pracy algorytmu genetycznego jest niezwykle istotny, gdyż bardzo łatwo potrafi on wpaść w pułapkę lokalnego optimum niebędącego jednak rozwiązaniem satysfakcjonującym. W takim przypadku najlepiej dopasowany osobnik rozprzestrzenia błyskawicznie swój chromosom w puli rodziców i blokuje eksplorację pozostałej przestrzeni rozwiązań,
- w celu zapobieżenia blokowaniu się rozwiązania w lokalnych ekstremach należałoby dodać do aplikacji metody dynamicznej kontroli parametrów oparte na sprawdzaniu wartości funkcji przystosowania na przestrzeni ostatnich pokoleń.

Możliwości rozwoju aplikacji:

- w bieżącym sposobie kodowania genów:
  - dodanie kolejnych metod selekcji rodziców,
  - dodanie kolejnych metod krzyżowania rodziców.
- zmiana sposobu kodowania genów - można podejść do zapisu rozwiązania w sposób odmienny i w genach zakodować istniejące metody częściowego logicznego rozwiązywania plansz sudoku (np. wyszukiwanie pól z najmniejszą ilością liczb potencjalnie możliwych do wprowadzenia), pozostawiając część stochastyczną na wypadek sytuacji, w której nie ma już możliwości zastosowania metod logicznych. Wtedy krzyżowanie rodziców zmieniałoby sposób operowania metodami wyszukiwania pól do ustawienia, dążąc do osobnika, w którym kolejność stosowania tych metod byłaby optymalna.

Podsumowując niniejszą pracę można stwierdzić, że metody oparte na algorytmach genetycznych są ciekawą alternatywą dla rozwiązywania problemów algorytmami deterministycznymi, jednak wymagają sporo dodatkowej uwagi przy kontroli uzyskiwanych danych pośrednich i dopasowywaniu na ich podstawie parametrów pracy algorytmu.

# Bibliografia

## Publikacje drukowane

- A01 Felgenhauer B., Jarvis F., **Mathematics of Sudoku I**, "Mathematical Spectrum" 2006, nr 39 (1), s. 15-22,  
<https://citeseerx.ist.psu.edu/document?repid=rep1&type=pdf&doi=d2b0eb07e7fa8bc5e7bb2cc24877e26db19fb2c2> (dostęp z dnia 04.05.2025)
- A02 Wnuk E., Łukasik E., **Analiza porównawcza algorytmów rozwiązujących Sudoku**, "Journal of Computer Sciences Institute" 2016, nr 2, s. 140-143,  
<https://ph.pollub.pl/index.php/jcsi/article/view/130> (dostęp z dnia 04.05.2025)
- A03 Grządko Ł., **Efektywny algorytm Knutha dla problemu Dokładnego Pokrycia i jego zastosowanie w trudnych łamigłówkach**, "Programista" 2021, nr 95,  
[https://programistamag.pl/wp-content/uploads/downloads/Programista\\_96\\_Algorytm\\_Knutha.pdf](https://programistamag.pl/wp-content/uploads/downloads/Programista_96_Algorytm_Knutha.pdf) (dostęp z dnia 04.05.2025)
- A04 David E. Goldberg, **Algorytmy genetyczne i ich zastosowanie**, WNT, Warszawa 2011
- A05 Tomasz D. Gwiazda, **Algorytmy genetyczne. Kompendium**, t. 1, PWN, Warszawa 2009
- A06 Tomasz D. Gwiazda, **Algorytmy genetyczne. Kompendium**, t. 2, PWN, Warszawa 2007
- A07 Zbigniew Michalewicz, **Algorytmy genetyczne + struktury danych = programy ewolucyjne**, WNT, Warszawa 2003
- A08 Danuta Rutkowska, **Sieci neuronowe, algorytmy genetyczne i systemy rozmyte**, PWN, Warszawa 1997

## Strony internetowe

- B01 *Sudoku*, portal Wikipedia, <https://pl.wikipedia.org/wiki/Sudoku> (dostęp z dnia 04.05.2025)
- B02 Jak grać w sudoku. Szczegółowy przewodnik, portal Escape Sudoku,  
<https://escape-sudoku.com/pl/blog/how-to-play-sudoku-detailed-guide> (dostęp z dnia 04.05.2025)
- B03 Poradnik sudoku, <https://www.sudoku.org.pl/> (dostęp z dnia 04.05.2025)
- B04 Lanterna, portal GitHub, <https://github.com/mabe02/lanterna> (dostęp z dnia 16.05.2025)
- B05 Introduction to Lanterna, <https://www.baeldung.com/java-lanterna> (dostęp z dnia 16.05.2025)

B06 Package com.googlecode.lanterna.terminal.swing,

<https://mabe02.github.io/lanterna/apidocs/3.0/com/googlecode/lanterna/terminal/swing/package-summary.html> (dostęp z dnia 18.05.2025)