***Wojskowa Akademia Techniczna***

***im. Jarosława Dąbrowskiego***



**Wydział Cybernetyki, kierunek informatyka - inżynieria systemów**

Realizacja projektu w ramach przedmiotu:

*Techniki Algorytmiczne*

Temat projektu (numer 15):

***Problem spełnialności***

***Algorytm genetyczny***

**Opracował:** Radosław Relidzyński, **Grupa:** WCY23IX3S4

Spis treści

[Problem spełnialności 3](#_Toc167881378)

[Algorytm genetyczny 3](#_Toc167881379)

[Rozwiązywanie sudoku jako problemu spełnialności 5](#_Toc167881380)

[Opis algorytmu dokładnego 7](#_Toc167881381)

[Opis algorytmu genetycznego 8](#_Toc167881382)

[Struktura implementacji 9](#_Toc167881383)

[Dostępny interfejs do uruchamiania programu 10](#_Toc167881384)

[Testy 10](#_Toc167881385)

[Badanie wydajności algorytmów 10](#_Toc167881386)

[Skrypt z programem interaktywnym dla użytkownika 11](#_Toc167881387)

[Pomiar złożoności czasowej dla badania wydajności algorytmów 11](#_Toc167881388)

[Pomiar złożoności pamięciowej dla badania wydajności algorytmów 12](#_Toc167881389)

[Sposób oceny jakości rozwiązania 13](#_Toc167881390)

[Sprawdzanie błędów w komórkach (czy jest to cyfra od 1 do 9) 13](#_Toc167881391)

[Sprawdzanie błędów w wierszach 13](#_Toc167881392)

[Sprawdzanie błędów w kolumnach 13](#_Toc167881393)

[Sprawdzanie błędów w kwadratach 3x3 14](#_Toc167881394)

[Podsumowanie sprawdzania przez sumowanie błędów 14](#_Toc167881395)

[Sprawdzanie samej poprawności z pominięciem zliczania błędów 14](#_Toc167881396)

[Szacowanie dla algorytmu genetycznego 14](#_Toc167881397)

[Szacowanie teoretycznej pesymistycznej złożoności obliczeniowej 15](#_Toc167881398)

[Szacowanie teoretycznej pesymistycznej złożoności pamięciowej 16](#_Toc167881399)

[Szacowanie teoretycznej oczekiwanej złożoności obliczeniowej 17](#_Toc167881400)

[Szacowanie teoretycznej oczekiwanej złożoności pamięciowej 17](#_Toc167881401)

[Szacowanie teoretycznej wrażliwości pesymistycznej i oczekiwanej 17](#_Toc167881402)

[Szacowanie teoretycznej dokładności 18](#_Toc167881403)

[Szacowanie dla algorytmu dokładnego 18](#_Toc167881404)

[Tworzenie wzoru do szacowania złożoności obliczeniowej 18](#_Toc167881405)

[Szacowanie teoretycznej pesymistycznej złożoności obliczeniowej 18](#_Toc167881406)

[Szacowanie teoretycznej pesymistycznej złożoności pamięciowej 18](#_Toc167881407)

[Szacowanie teoretycznej oczekiwanej złożoności obliczeniowej 18](#_Toc167881408)

[Szacowanie teoretycznej oczekiwanej złożoności pamięciowej 18](#_Toc167881409)

[Szacowanie teoretycznej wrażliwości pesymistycznej i oczekiwanej 18](#_Toc167881410)

[Szacowanie teoretycznej dokładności 18](#_Toc167881411)

[Przeprowadzenie eksperymentów 18](#_Toc167881412)

# Problem spełnialności

**Problem spełnialności** – „zagadnienie rachunku zdań, określające czy dla danej formuły logicznej istnieje takie podstawienie (wartościowanie) zmiennych zdaniowych, żeby formuła była prawdziwa. Jest ono równoważne negacji odpowiedzi na pytanie czy negacja tej formuły jest tautologią.”  
[<https://pl.wikipedia.org/wiki/Problem_spe%C5%82nialno%C5%9Bci>]

Źródła Youtube:

* <https://www.youtube.com/watch?v=-wlUDJZb6-Q>
* <https://www.youtube.com/watch?v=4K1MyG4ljI8>

Problem spełnialności dotyczy formuł w postaci CNF (Conjunctive Normal Form), czyli w postaci koniunkcji (mnożenia) klauzul będących alternatywą (sumą) literałów (zmiennych boolowskich z negacją lub bez). Przykład takiej formuły:

Problem SAT jest problemem decyzyjnym, określamy przy jego pomocy to, czy istnieje jakiekolwiek przypisanie wartości do zmiennych, które spełnia formułę.

Problem spełnialności należy do klasy problemów NP-pełnych (tak samo trudnych jak każdy problem w klasie NP).

Klasa NP (niedeterministyczny wielomianowy czas) obejmuje problemy decyzyjne, dla których każda instancja (np. zbiór wartości) powodująca prawdziwość formuły może być zweryfikowany w czasie wielomianowym przez deterministyczną maszynę Turinga.

Przykładowe zastosowania problemu SAT:

* Weryfikacja poprawności układów scalonych
* Poszukiwanie optymalnej sekwencji działań w robotyce
* Analiza problemów strukturalnych i funkcjonalnych w sekwencji DNA
* Projektowanie sieci neuronowych w uczeniu maszynowym

# Algorytm genetyczny

**Algorytm genetyczny** – „rodzaj heurystyki przeszukującej przestrzeń alternatywnych rozwiązań problemu w celu wyszukania najlepszych rozwiązań.”  
[<https://pl.wikipedia.org/wiki/Algorytm_genetyczny>]

Źródła Youtube:

* <https://www.youtube.com/live/ilq7A0ZXsGY?si=GvElsyltLUH8Pm4x>
* <https://www.youtube.com/watch?v=SMYNxNiABmU>
* <https://www.youtube.com/watch?v=MwnRzXj8pJA>

Algorytm genetyczny pozwala na efektywne znalezienie dobrych rozwiązań, natomiast nie gwarantuje znalezienia najlepszego rozwiązania.

Działanie algorytmu krok po kroku:

1. Inicjalizacja populacji – stworzenie zbioru początkowej populacji rozwiązań (osobników).
2. Ocena – zmierzenie przy pomocy funkcji przystosowania jak dobrze dany osobnik rozwiązuje problem
3. Selekcja – wybór osobników do reprodukcji na podstawie podanej oceny (im wyższa ocena, tym bardziej są faworyzowane)
4. Krzyżowanie (rekombinacja) – dobieranie osobników w pary i mieszanie ich danych genetycznych w celu utworzenia nowych osobników
5. Mutacja – wprowadzanie niewielkich losowych zmian w genach osobników w celu zapobiegnięcia konwergencji (upodobnienia wszystkich osobników).
6. Nowa generacja – zastąpienie zainicjalizowanej populacji nową będącą wynikiem dotychczasowych operacji.
7. Kryterium stopu – algorytm jest powtarzany do momentu spełnienia kryterium stopu (dla odpowiedniego przystosowania, maksymalnej liczby generacji, warunku zakończenia procesu)

Przykładowe zastosowania algorytmu genetycznego:

* Projektowanie skomplikowanych systemów (mosty, układy elektroniczne) pod wybrane wymagania minimalne
* Optymalizacja tras przewozowych w przemyśle transportowym
* Analiza i modelowanie sekwencji DNA w celu odkrywania leków lub zrozumienie mechanizmu chorób

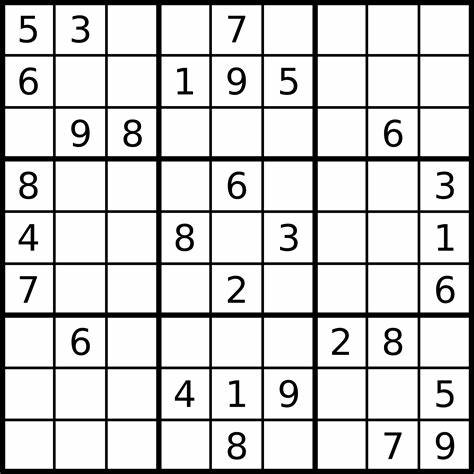
Podstawowe definicje i problem komiwojażera:

1. <https://home.agh.edu.pl/~vlsi/AI/gen_t/>

# Rozwiązywanie sudoku jako problemu spełnialności

**Sudoku** – „łamigłówka, której 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” lub „podkwadratami”) znalazło się po jednej cyfrze od 1 do 9.”  
[<https://pl.wikipedia.org/wiki/Sudoku>]

Przykładowe zadanie sudoku:



Poprawnie skonstruowane sudoku zakłada istnienie tylko jednego prawidłowego rozwiązania, co czyni je odpowiednim zadaniem dla problemu SAT. Na podstawie definicji można również stworzyć formę CNF, która przedstawia sprawdzenie poprawności proponowanego rozwiązania jako problem SAT. Do osiągnięcia tego należy opisać takie warunki:

1. W każdej komórce musi być jakaś wartość od 1 do 9.
2. W każdej komórce nie może być więcej niż jedna wartość.
3. W każdym wierszu nie może się powtórzyć żadna wartość.
4. W każdej kolumnie nie może się powtórzyć żadna wartość.
5. W każdym wyznaczonym kwadracie 3x3 nie może powtórzyć się żadna wartość.

Zanim jeszcze nastąpi tworzenie wzoru reprezentującego problem sudoku jako problem SAT należy dodać ważną na rzecz implementacji informację o minimalniej koniecznej liczbie wypełnionych komórek w problemie, które powodują, że sudoku może zostać rozwiązane:

„Jeśli liczba podanych cyfr w diagramie jest mniejsza od 17, to łamigłówka nie ma jednoznacznego rozwiązania”.  
[<https://pl.wikipedia.org/wiki/Sudoku#Liczba_mo%C5%BCliwych_plansz>]

Zatem największa możliwa liczba pustych komórek to

Do wyrażenia tego w postaci CNF wykorzystam symbol koniunkcji (⋀) dzięki któremu będzie można przy pomocy prostych formuł opisać wszystkie potrzebne warunki. Zapis każdego literału będzie następujący:

Oznacza to, że dla danej komórki będziemy sprawdzać, czy posiada ona (lub czy nie posiada dla negacji) podaną wartość. Jeśli posiada to ma zwrócić wartość prawda, jeśli nie to fałsz (lub na odwrót dla negacji).

Zapis tych warunków będzie następujący:

1. W każdej komórce musi być jakaś wartość od 1 do 9.
2. W każdej komórce nie może być więcej niż jedna wartość.
3. W każdym wierszu nie może się powtórzyć żadna wartość.
4. W każdej kolumnie nie może się powtórzyć żadna wartość.
5. W każdym wyznaczonym kwadracie 3x3 nie może powtórzyć się żadna wartość.

Wykaz kolejnych zmiennych do indeksowania w warunku dla kwadratów (punkt 5):

1. Podział na małe kwadraty:
   * m – numer rzędu kwadratów numerowane od 0
   * n – numer kolumny kwadratów numerowany od 0
2. Poruszanie się w małych kwadratach:
   * i – wiersz małego kwadratu 3x3
   * j – kolumna małego kwadratu 3x3
3. Wyznaczanie wartości do sprawdzania obecności w komórkach:
   * k – wartość komórki
4. Porównywanie wartości komórki z wartościami innych komórek:
   * p – wiersz porównywanej komórki w małym kwadracie
   * q – kolumna porównywanej komórki w małym kwadracie

Aby uznać rozwiązanie za poprawne muszą zostać spełnione wszystkie powyższe warunki, czyli ostateczny wzór wygląda tak:

Ten wzór stanowi sprawdzenie czy rozwiązanie dla sudoku jest poprawne, będzie to potrzebne do implementacji rozwiązania.

Źródła:

* <https://sat.inesc-id.pt/~ines/publications/aimath06.pdf>
* <https://users.aalto.fi/~tjunttil/2020-DP-AUT/notes-sat/solving.html>
* <https://www.cs.cmu.edu/~hjain/papers/sudoku-as-SAT.pdf>

# Opis algorytmu dokładnego

Jako algorytm dokładny wykorzystam iterację między wszystkimi możliwymi rozwiązaniami sudoku. Oto kolejne działania, jakie zastosuję:

1. Wstaw w każdą wolną komórkę wartość 1.
2. Sprawdź poprawność rozwiązania.
3. Jeśli rozwiązanie jest poprawne, zwróć wynik i zakończ działanie.
4. Jeśli wartość pierwszej uzupełnianej komórki nie wynosi 9, zwiększ jej wartość o 1
5. Jeśli wartość pierwszej uzupełnianej komórki wynosi 9 wykonaj następujące działanie:
   1. Nadaj bieżącej komórce wartość 1
   2. Wykonaj zmianę wartości na kolejnych komórkach. Dopóki wartość analizowanej komórki nie będzie różna od 9 wykonuj te kroki:
      * Jeśli wartość analizowanej komórki jest różna od 9 zwiększ jej wartość o 1 i zakończ działanie kroku 5b
      * Jeśli wartość komórki wynosi 9 zmień jej wartość na 1
      * Przejdź do następnej uzupełnianej komórki
6. Weź nowo utworzone rozwiązanie i wykonaj na nim kolejne punkty zaczynając od kroku 2.

Jest to bardzo nieoptymalne rozwiązanie powodujące, że szukamy po wszystkich możliwych rozwiązaniach, czyli będzie to pula wielkości 9n, gdzie n to liczba komórek do uzupełnienia.

# Opis algorytmu genetycznego

Algorytm genetyczny będzie przeszukiwał możliwe rozwiązania do momentu osiągnięcia kryterium stopu, które na rzecz rozwiązywania sudoku będzie w momencie znalezienia poprawnego rozwiązania.

Oto kolejne kroki służące do realizacji algorytmu genetycznego dla rozwiązywania sudoku:

1. Tworzenie populacji – generowanie osobników w postaci losowych rozwiązań. Liczba generowanych rozwiązań będzie możliwa do skonfigurowania
2. Ocena – wyznaczanie za pomocą funkcji przystosowania jak dobrze dany osobnik rozwiązuje problem. Funkcja przystosowania będzie określała jakość rozwiązania na podstawie liczby błędów znalezionych dla każdego wiersza, kolumny i kwadratu 3x3 (im więcej błędów, tym mniejsza jakość rozwiązania).
3. Selekcja – wybór osobników do reprodukcji na podstawie oceny. Każda selekcja będzie miała maksymalną liczbę osobników dopuszczonych do następnej iteracji.
4. Krzyżowanie – dobieranie osobników w pary i mieszanie ich danych w celu utworzenia nowych osobników. Będzie to na zmianę pobieranie kolejnych wartości od dwójki rodziców do dziecka.
5. Mutacja – niewielka zmiana w genach osobników, każdy osobnik w populacji będzie miał zmienioną wartość jednej losowej komórki.
6. Nowa generacja – zastąpienie starej populacji nową
7. Kryterium stopu – jak wspomniano wcześniej, algorytm kończy swoje działanie w momencie znalezienia poprawnego rozwiązania, czyli takiego, którego liczba błędów znalezionych przez funkcję przystosowania wynosi 0.

W zależności od początkowej populacji liczba rozwiązań do sprawdzenia może się znacznie różnić, ale odpowiednia implementacja algorytmu genetycznego pozwoli na szybkie dotarcie do obszaru rozwiązań zawierającego to odpowiednie, co w konsekwencji da nam wynik przy analizie uśredniając znacznie mniejszej liczby rozwiązań niż przy algorytmie dokładnym.

# Struktura implementacji

Struktura projektu:

A screenshot of a computer

Description automatically generated

Opis każdego elementu:

1. „good\_solutions” – zbiór plików tekstowych zawierających w pełni rozwiązane sudoku do generowania losowych problemów do rozwiązania
2. „performance\_results” – zbiór plików typu Excel będących podsumowaniami uruchomień testów wydajności ze skryptu „run\_performance.py”
3. „problems\_source” – zbiór plików tekstowych zawierających nierozwiązane sudoku, użytkownik może umieścić tu swój problem i przy pomocy skryptu „interactive\_sudoku.py” otrzymać rozwiązanie
4. „tests” – zbiór podstawowych testów badających czy mechanizm sudoku oraz algorytmu zostały dobrze zaimplementowane
5. „exact\_algorithm.py” – skrypt z implementacją algorytmu dokładnego do rozwiązywania sudoku
6. „genetic\_algorithm.py” - skrypt z implementacją algorytmu genetycznego do rozwiązywania sudoku
7. „interactive\_sudoku.py” – skrypt umożliwiający użytkownikowi uruchomienie narzędzia do rozwiązywania sudoku
8. „run\_performance.py” – skrypt przygotowujący zbiór problemów sudoku do rozwiązania, zbierający wyniki wydajności pod względem czasu wykonania oraz wykorzystania pamięci. Zebrane wyniki następnie zestawia w formie zbioru wykresów i zapisuje do pliku typu Excel
9. „source\_manager.py” – skrypt pomocniczy, służący do zapisywania oraz pobierania problemów sudoku lub dobrych rozwiązań, zarządza przeznaczonymi do tego plikami tekstowymi.
10. „Sudoku.py” – kluczowy skrypt zawierający klasę Sudoku przechowującą wszystkie informacje o danym problemie. Zawiera również zaimplementowany algorytm sprawdzający poprawność rozwiązania (razem z funkcją przystosowania dla algorytmu genetycznego)

# Dostępny interfejs do uruchamiania programu

### Testy

Zbiór testów jest następujący:

A screenshot of a computer

Description automatically generated

Są to proste testy służące tylko do tego, żeby sprawdzić poprawność implementacji algorytmów. W trakcie implementacji służyły jako wyznacznik, kiedy jest gotowa (na zasadzie Test-Driven Development).

### Badanie wydajności algorytmów

Do tego służy skrypt „run\_performance.py”, dzięki któremu można określić różne parametry uruchomienia, a następnie program sam zajmie się ich uruchomieniem, zapisaniem i wstępną analizą.

Tak wygląda zbiór dostępnych konfiguracji, które z poziomu deklaracji można konfigurować:

NUM\_OF\_SOLUTIONS\_TO\_USE = 3  
ATTEMPTS\_AMOUNT = 5  
GENETIC\_ALGORYTHM\_POPULATION\_NUMBERS = [5, 10, 20]  
NUM\_OF\_EMPTY\_CELLS\_RANGE = range(3, 7)  
OUTPUT\_FILE\_NAME = 'performance\_results.xlsx'  
  
runner = PerformanceRunner(NUM\_OF\_SOLUTIONS\_TO\_USE, ATTEMPTS\_AMOUNT, NUM\_OF\_EMPTY\_CELLS\_RANGE,  
 GENETIC\_ALGORYTHM\_POPULATION\_NUMBERS, OUTPUT\_FILE\_NAME)  
runner.main()

Opis każdego parametru:

1. „NUM\_OF\_SOLUTIONS\_TO\_USE” – liczba dobrych rozwiązań na podstawie których generowane będą problemy dla algorytmów.
2. „NUM\_OF\_EMPTY\_CELLS\_RANGE” – zakres różnych liczb pustych komórek w problemach sudoku.
3. „GENETIC\_ALGORYTHM\_POPULATION\_NUMBERS” – zbiór wielkości populacji dla różnych uruchomień algorytmu genetycznego.
4. „ATTEMPTS\_AMOUNT” – liczba podejść do każdego pojedynczego problemu.
5. „OUTPUT\_FILE\_NAME” – nazwa pliku, do którego zapisane zostaną wyniki.

Liczba uruchomień algorytmu wyraża się takim wzorem:

Wynikiem działania programu jest plik excel zawierający następujące elementy:

1. Tablicę zawierającą zmierzony czas wykonania i wykorzystanie pamięci przez algorytm.
2. Wykresy zestawiające czas wykonania lub wykorzystania pamięci w zależności od typu algorytmu (dokładny lub genetyczny dla danej wielkości populacji).
3. Tablicę zawierającą średnie czasy wykonania lub wykorzystania pamięci.
4. Wykresy zestawiające średni czas wykonania lub średnie wykorzystanie pamięci w zależności od typu algorytmu (dokładny lub genetyczny dla danej wielkości populacji).

### Skrypt z programem interaktywnym dla użytkownika

Jest to skrypt „interactive\_sudoku.py”, dzięki niemu użytkownik może zadać dany problem do rozwiązania. Dostarcza go w formie pliku tekstowego zamieszczając w odpowiednim miejscu. Następnie program pyta o nazwę tego pliku oraz o wielkość populacji dla algorytmu genetycznego i następnie uruchamia obydwa algorytmu wypisując czas wykonania jednego i drugiego.

# Pomiar złożoności czasowej dla badania wydajności algorytmów

Poniżej przedstawiony jest fragment kodu pokazujący sposób badania czasu wykonania algorytmu:

sudoku\_problem = get\_problem(sudoku\_problem\_file)  
sudoku = Sudoku(sudoku\_problem)  
  
print("Solving with genetic algorithm")  
genetic\_algorithm = GeneticAlgorithm(sudoku)  
  
start\_time = time.time()  
  
result = genetic\_algorithm.genetic\_algorithm()  
  
end\_time = time.time()  
elapsed\_time = end\_time - start\_time

if not result.is\_valid():  
 raise Exception("Genetic algorithm result is not valid")  
print(f'Solved with genetic algorithm with time {elapsed\_time}')

Jak widać program pobiera znacznik czasowy przed i po wywołaniu algorytmu, a następnie podaje różnicę pomiędzy nimi. Dzięki temu znany jest czas wykonania.

# Pomiar złożoności pamięciowej dla badania wydajności algorytmów

Poniżej zamieszczona jest implementacja mechanizmu uruchomienia sprawdzania wydajności algorytmu genetycznego z pomiarem wykorzystania pamięci:

@staticmethod  
def run\_genetic\_algorithm\_performance(good\_solution, num\_of\_empty\_cells, population\_number):  
 *""" genetic algorithm runner with time and memory measure"""* print(f"Running GeneticAlgorithm for p={population\_number} and n={num\_of\_empty\_cells}")  
  
 def run\_algorithm():  
 *""" run genetic\_algorithm """* sudoku = get\_random\_sudoku(num\_of\_empty\_cells, good\_solution)  
 genetic\_algorithm = GeneticAlgorithm(sudoku, population\_number=population\_number)  
  
 start\_time = time.time()  
  
 result = genetic\_algorithm.genetic\_algorithm()  
  
 end\_time = time.time()  
  
 return result, end\_time - start\_time  
  
 memory\_usage\_info, run\_result = memory\_usage(run\_algorithm, interval=0.1, max\_usage=True, retval=True)  
  
 result\_sudoku, elapsed\_time = run\_result  
  
 if not result\_sudoku.is\_valid():  
 raise ValueError(f"GeneticAlgorithm {population\_number}: Sudoku not solved properly")  
  
 print(f"Finished GeneticAlgorithm for p={population\_number} and n={num\_of\_empty\_cells} "  
 f"with time {elapsed\_time} seconds and max memory usage {memory\_usage\_info} MiB")  
 return elapsed\_time, memory\_usage\_info

Do pomiaru wykorzystania pamięci zastosowane jest narzędzie „memory\_usage” z biblioteki „memory\_profiler”. Jest to funkcja badająca zużycie pamięci w czasie, zwracająca listę pomiarów zużycia pamięci w czasie działania programu. W przypadku powyższego zastosowania flaga „max\_usage” jest ustawiona jako prawda, więc funkcja zwraca jedynie wartość największego pomiaru.

W systemie Windows „memory\_profiler” wykorzystuje funkcję systemową o nazwie „GetProcessMemoryInfo”, która zwraca zużycie pamięci dla podanego procesu, stąd funkcja wie, ile pamięci zostało zużyte w danym czasie.

Źródła wiedzy o bibliotece:

* <https://pypi.org/project/memory-profiler/>
* <https://www.youtube.com/watch?v=tIKo2uex8x4>
* <https://www.youtube.com/watch?v=3PdmLQIZpwE>
* <https://learn.microsoft.com/en-us/windows/win32/api/psapi/nf-psapi-getprocessmemoryinfo>

# Sposób oceny jakości rozwiązania

Implementacja oceny jakości rozwiązania wynika bezpośrednio ze zdefiniowanego wzoru w formacie CNF definiującej problem sudoku jako problem spełnialności (SAT). Implementacja wszystkich warunków znajduje się wewnątrz klasy „Sudoku”. Poniżej prezentuję kolejne metody sprawdzające zdefiniowane wcześniej warunki:

### Sprawdzanie błędów w komórkach (czy jest to cyfra od 1 do 9)

def is\_solved(self) -> bool:  
 *""" check if whole board is filled with digits from 1 to 9 """* return all(self.board[i][j] in range(1, 10) for i in range(9) for j in range(9))

### Sprawdzanie błędów w wierszach

def \_\_count\_mistakes\_in\_rows(self, only\_validation=False) -> int:  
 mistake\_count = 0  
 for row in self.board:  
 values\_seen = set()  
 for value in row:  
 if value in values\_seen:  
 if only\_validation:  
 return -1  
 mistake\_count += 1  
 values\_seen.add(value)  
 return mistake\_count

### Sprawdzanie błędów w kolumnach

def \_\_count\_mistakes\_in\_columns(self, only\_validation=False) -> int:  
 mistake\_count = 0  
 for col\_index in range(9):  
 values\_seen = set()  
 for row\_index in range(9):  
 value = self.board[row\_index][col\_index]  
 if value in values\_seen:  
 if only\_validation:  
 return -1  
 mistake\_count += 1  
 values\_seen.add(value)  
 return mistake\_count

### Sprawdzanie błędów w kwadratach 3x3

def \_\_count\_mistakes\_in\_squares(self, only\_validation=False) -> int:  
 mistake\_count = 0  
 for start\_row in range(0, 9, 3):  
 for start\_col in range(0, 9, 3):  
 values\_seen = set()  
 for i in range(3):  
 for j in range(3):  
 value = self.board[start\_row + i][start\_col + j]  
 if value in values\_seen:  
 if only\_validation:  
 return -1  
 mistake\_count += 1  
 values\_seen.add(value)  
 return mistake\_count

### Podsumowanie sprawdzania przez sumowanie błędów

def count\_mistakes(self) -> int:  
 *""" count number of mistakes in board for rows, cols and 3x3 squares """* if not self.is\_solved():  
 raise ValueError('Sudoku is not solved yet')  
 count\_mistakes\_rows = self.\_\_count\_mistakes\_in\_rows()  
 count\_mistakes\_cols = self.\_\_count\_mistakes\_in\_columns()  
 count\_mistakes\_squares = self.\_\_count\_mistakes\_in\_squares()  
 return count\_mistakes\_rows + count\_mistakes\_cols + count\_mistakes\_squares

### Sprawdzanie samej poprawności z pominięciem zliczania błędów

def is\_valid(self) -> bool:  
 *""" check if board is solved without returning the amount of errors """* if not self.is\_solved():  
 raise ValueError('Sudoku is not solved yet')  
 if self.\_\_count\_mistakes\_in\_rows(only\_validation=True) == -1:  
 return False  
 if self.\_\_count\_mistakes\_in\_columns(only\_validation=True) == -1:  
 return False  
 if self.\_\_count\_mistakes\_in\_squares(only\_validation=True) == -1:  
 return False  
 return True

Tak właśnie prezentuje się cały mechanizm sprawdzania poprawności. Służy on za równo do oceny czy sudoku jest poprawnie rozwiązane jak i stanowi funkcję przystosowania dla algorytmu genetycznego (ocena wynika z liczby błędów w rozwiązaniu).

# Szacowanie dla algorytmu genetycznego

Do szacowania wykorzystam te oznaczenia zmiennych:

### Szacowanie teoretycznej pesymistycznej złożoności obliczeniowej

Szacowanie to odbędzie się poprze kolejne szacowanie złożoności każdego kroku algorytmu genetycznego. Poniżej zaprezentowane są po kolei szacowania dla każdego kroku oraz podsumowanie całości:

#### Inicjalizacja populacji

W ramach inicjalizacji populacji zachodzi taki zestaw operacji:

1. Tworzy w pętli kolejne osobniki do populacji. **Złożoność tej operacji to będzie**  pomnożona razy złożoność obliczeniową każdej operacji wewnątrz niej.
2. Tworzy głęboką kopię problemu sudoku – tablicy 9x9, listy wypełnialnych komórek i atrybutu oceny jakościowej (na ten moment bez wartości). **Tutaj złożoność wynosi** .
3. W miejsce każdej wypełnialnej komórki wpisuje losową wartość od 1 do 9. **Złożoność tej operacji to**
4. Dodaje nowo utworzonego osobnika do listy. Złożoność tej operacji to

Podsumowując, złożoność obliczeniowa inicjalizacji populacji to

#### Ocena

Ocena to wywołanie w pętli oceny kolejnych osobników. Złożoność tej operacji to będzie pomnożona razy złożoność obliczeniową samej oceny, która w tym przypadku wynosi , gdyż nie zależy ona od żadnej zmiennej. Podsumowując, **złożoność ta wynosi**

#### Selekcja

W ramach selekcji następuje posortowanie populacji względem wielkości oceny, a następnie podstawienie pod listę osobników nowej listy, tym razem zawierającej p osobników z najlepszą oceną z listy pierwotnej. Sortowanie to złożoność , a ucinanie to złożoność , więc razem **złożoność ta to**

#### Krzyżowanie

Implementacja krzyżowania polega na losowym dobraniu osobników w pary i na podstawie każdej pary tworzenie dziecka będącego dobieraniem kolejnych wartości na zmianę od rodzica. Kolejne złożoności dla tych operacji:

1. Przetasowanie populacji –
2. Iteracja po każdej parze –
3. Kopiowanie głębokie sudoku –
4. Iteracja po każdej wypełnialnej komórce –
5. Przypisanie wartości od odpowiedniego rodzica –

Podsumowując, **złożoność będzie wynosiła**

#### Mutacja

Podczas mutacji każdy osobnik ma zmienioną jedną losową wartość, co daje złożoność

#### Nowa generacja

Nie zachodzi tu żadna nowa operacja, podmiana nowej generacji zachodzi na bieżąco w krokach 3-5 algorytmu genetycznego.

#### Kryterium stopu

Dla każdego rozwiązania w populacji sprawdza czy jest poprawne, jeśli jest to kończy działanie algorytmu, a jeśli nie to uruchamia ponownie kroki 2-6. Złożoność obliczeniowa to będzie liczba osobników razy złożoność obliczeniowa sprawdzania poprawności. Sprawdzania dla wierszy, kolumn, kwadratów 3x3 mają złożoności obliczeniowe , więc wykonując to sprawdzanie dla całej populacji **osiągamy złożoność**

#### Razem

Cały algorytm genetyczny to inicjalizacja populacji oraz wykonywanie w pętli pozostałych kroków do momentu spełnienia kryterium stopu. Zatem podsumowując całą teoretyczną złożoność, będzie ona wynosić:

### Szacowanie teoretycznej pesymistycznej złożoności pamięciowej

#### Złożoność pamięciowa klasy „Sudoku”

W klasie „Sudoku” są następujące atrybuty:

* board – tablica 9x9 z wartościami komórek
* non\_perm\_cells\_list – lista pustych komórek do znalezienia poprawnych wartości przez algorytm
* mark – ocena jakości rozwiązania

Zatem teoretyczna złożoność pamięciowa klasy „Sudoku” zależy wyłącznie od liczby pustych komórek, których liczba stanowi liczbę elementów listy „non\_perm\_cells\_list”. Więc jest to złożoność

#### Złożoność pamięciowa klasy „GeneticAlgorithm”

W klasie „Sudoku” są następujące atrybuty:

* sudoku\_to\_solve – obiekt klasy „Sudoku” z nierozwiązanym problemem
* population\_number – liczba określająca wielkość populacji
* population – lista reprezentująca populację, zmieniana w każdej iteracji algorytmu

Na teoretyczną złożoność pamięciową wpływ będzie miała wielkość populacji, ponieważ określa ona wielkość listy „population”. Będzie to więc to złożoność

#### Razem

Dla każdego członka populacji z klasy „GeneticAlgorithm” przypada jeden obiekt „Sudoku”. Stąd teoretyczna złożoność pamięciowa tej części implementacji to

### Szacowanie teoretycznej oczekiwanej złożoności obliczeniowej

Teoretyczna oczekiwana złożoność obliczeniowa jest równa pesymistycznej, czyli wynosi

Wynika to z faktu, że każdy krok algorytmu genetycznego, bez względu na warunki początkowe, ma ustaloną strukturę operacji, która prowadzi do stałej złożoności. Każdy z etapów działa w z góry określony sposób, a zmienne podawane do algorytmu mają znaczenie w przypadku liczby iteracji, która będzie losowa w zależności od tego jaka będzie wylosowana pierwsza populacja. Nie zmienia to jednak faktu, złożoność obliczeniowa jest stała zarówno rozważając złożoność pesymistyczną, jak i oczekiwaną.

### Szacowanie teoretycznej oczekiwanej złożoności pamięciowej

Teoretyczna oczekiwana złożoność pamięciowa jest równa pesymistycznej, czyli wynosi

Powodem takiego stanu rzeczy jest sposób implementacji algorytmu. Zawiera się on w 2 klasach, które mają jasno określoną strukturę danych, a operacje na listach w nich zawartych nie przekraczają konkretnej wartości. Stąd niezależnie od wielkości zmiennych p oraz n, zarówno pesymistyczna jak i oczekiwana złożoność pamięciowa będzie stała.

### Szacowanie teoretycznej wrażliwości pesymistycznej i oczekiwanej

Jak zostało uzasadnione wcześniej, teoretyczna złożoność obliczeniowa pesymistyczna i oczekiwana jest taka sama, co oznacza, że algorytm jest **mało wrażliwy czasowo**. Jednak nie jest on w pełni niewrażliwy, ponieważ wpływa na niego jakość pierwszej populacji.

W przypadku teoretycznej wrażliwości pesymistycznej, będzie ona w momencie, gdy osobniki w populacji są bardzo daleko od poprawnego rozwiązania w przestrzeni rozwiązań.

W przypadku wrażliwości oczekiwanej będzie to przypadek, kiedy nie będzie przeważającej liczby rozwiązań zarówno w dużej, jak i małej odległości od rozwiązania poprawnego.

### Szacowanie teoretycznej dokładności

Ze względu na zastosowane kryterium stopu, implementacja algorytmu genetycznego ma na celu znalezienie poprawnego rozwiązania, zatem jest on dokładny.

# Szacowanie dla algorytmu dokładnego

Do szacowania wykorzystam tą zmienną:

### Szacowanie teoretycznej pesymistycznej złożoności obliczeniowej

Algorytm dokładny wypełnia brakujące komórki jedynkami, a następnie kolejno zwiększa wartość komórek do momentu znalezienia poprawnego rozwiązania. Oznacza to, że sprawdza wszystkie możliwe kombinacje wartości od 1 do 9 dla każdej pustej komórki. Razem daje to liczbę 9n. Dodatkowe operacje wewnątrz implementacji mają złożoność . W rezultacie teoretyczna złożoność obliczeniowa to **.**

### Szacowanie teoretycznej pesymistycznej złożoności pamięciowej

Na podstawie szacowań z algorytmu genetycznego wiadomo, że klasa „Sudoku” ma złożoność pamięciową **.** Klasa „ExactAlgorithm” zawiera problem sudoku oraz zmienną „non\_perm\_cell\_index”, która nie ma wpływy na teoretyczną złożoność pamięciową. Stąd teoretyczna złożoność pamięciowa algorytmu dokładnego wynosi **.**

### Szacowanie teoretycznej oczekiwanej złożoności obliczeniowej

Teoretyczna oczekiwana złożoność obliczeniowa jest równa pesymistycznej, czyli wynosi

Wynika to z faktu, że algorytm genetyczny, bez względu na warunki początkowe, ma ustaloną strukturę operacji, która prowadzi do stałej złożoności. Niezależnie od problemu iteracja będzie przebiegać tak samo, przez co złożoność obliczeniowa jest stała zarówno dla złożoności pesymistycznej, jak i oczekiwanej.

### Szacowanie teoretycznej oczekiwanej złożoności pamięciowej

Teoretyczna oczekiwana złożoność pamięciowa jest równa pesymistycznej, czyli wynosi **.**

Algorytm dokładny nie jest wymagający pamięciowo, jego główne dane to informacja o problemie, co powoduje, że jest jego teoretyczna złożoność pamięciowa jest stała.

### Szacowanie teoretycznej wrażliwości pesymistycznej i oczekiwanej

Jak w przypadku algorytmu genetycznego, teoretyczna złożoność obliczeniowa pesymistyczna i oczekiwana jest taka sama, co oznacza, że algorytm jest **mało wrażliwy czasowo**. Natomiast istnieją takie przypadki, kiedy algorytm dokładny będzie miał znacznie większy czas działania.

Wrażliwość algorytmu dokładnego wynika bezpośrednio z tego jaki problem zostanie zadany. Algorytm przechodzi po wszystkich rozwiązaniach zaczynając od wstawienia samych jedynej, więc im większa wartość w komórce tym więcej iteracji zostanie dokonanych. Dodatkowo znaczenie ma pozycja komórki – im bliżej pierwszego wiersza lub pierwszej kolumny, tym większy wpływ będzie to miało na wrażliwość algorytmu.

Stąd pesymistyczną wrażliwość można zaobserwować dla przypadku, kiedy jest 9 pustych komórek i każda z nich będzie miała poprawną wartość 9. Wtedy będzie to dokładnie 9n rozwiązań do sprawdzenia.

Oczekiwana wrażliwość będzie wtedy, kiedy rozwiązań do sprawdzenia będzie 9n/2, co zajdzie dla przykładowego przypadku, kiedy 9 pustych komórek posiada poprawną wartość 5.

### Szacowanie teoretycznej dokładności

Algorytm dokładny, mimo że nie optymalny, zawsze zwraca poprawny wynik, jest więc algorytmem dokładnym

# Przeprowadzenie eksperymentów

W ramach eksperymentów jest zaimplementowany skrypt uruchamiający różne konfiguracje algorytmów i zestawiający je w pliku Excel. Poniżej przedstawiam przykładowe zestawienie z wykresami dla każdego uruchomienia oraz dla średnich wartości.

A group of people on a white background

Description automatically generated

A group of graphs on a white background

Description automatically generated

A screenshot of a graph

Description automatically generated

Wykres porównaniu czasu pokazuje precyzyjnie, że wraz ze wzrostem liczby komórek do uzupełnienia w sudoku, czas wykonania algorytmu dokładnego staje się coraz większy od czasu wykonania algorytmu genetycznego.

Jeśli chodzi o złożoność pamięciową, różnice są nieznaczne, można zaobserwować, że w zależności od wielkości populacji algorytm genetyczny zajmuje nieco więcej pamięci.

Dodatkowo, poniżej przedstawiony jest wykres dla specjalnego uruchomienia samego algorytmu dokładnego dla różnej liczby pustych komórek, widać tutaj wzrost wykładniczy czasu wykonania.

Została również podjęta próba dla 16 pustych komórek, gdzie czas algorytmu genetycznego wyniósł 153 sekundy, a algorytm dokładny po dwóch godzinach nie był w stanie zwrócić wyniku, co dobrze obrazuje różnicę czasową oraz jej złożoności.

# Wnioski

W ramach zadania laboratoryjnego udało się skutecznie zaimplementować i zbadać algorytm genetyczny w ramach problemu sudoku ujętego jako problem spełnialności. Przeprowadzona analiza pokazuje skuteczność algorytmu genetycznego przy rozwiązywaniu złożonych problemów.

Algorytm genetyczny pozwala na efektywne przeszukiwanie przestrzeni rozwiązań, zaczynając od losowych wartości, przybliżając się z każdą iteracją do satysfakcjonującego rozwiązania. Nie w każdym przypadku jest to rozwiązanie najlepsze, natomiast przy odpowiednim kryterium sukcesu można osiągnąć bardzo dobry wynik.

Algorytm dokładny powstał, aby kontrastując z algorytmem genetycznym pokazać jak ważne jest odpowiednie podejście do przeszukiwania rozwiązań tak, aby przy jak najmniejszej liczbie rozwiązań znaleźć to satysfakcjonujące.

Podsumowując, algorytm genetyczny okazał się bardzo efektywny w rozwiązywaniu sudoku jako problemu spełnialności. Jego zdolność do szybkiego przeszukiwania przestrzeni rozwiązań oraz niewielkie zużycie pamięci uzasadnia jego dużą popularność. Algorytm dokładny, choć teoretycznie zawsze zapewnia poprawne rozwiązanie, to ze względu na swoją wysoką złożoność czasową jest znacznie mniej preferowanym narzędziem w rozwiązywaniu sudoku.