Rozwiązywanie problemu optymalizacyjnego w grze sudoku.

Gracjan Kościński

Czerwiec 2024

Spis treści

1 Wstęp

Sudoku to łamigłówka logiczna, która polega na wypełnieniu 9x9 siatki cyframi od 1 do 9 w taki sposób, aby każda cyfra pojawiała się tylko raz w każdym wierszu, kolumnie oraz w każdej z dziewięciu 3x3 podsiatek (nazywanych także „blokami" lub „regionami"). W poniższej pracy przedstawię porównanie algorytmów rozwiązujących problem gry w sudoku (Backtracking, algorytm genetyczny oraz PSO) dla różnego stopnia skomplikowania gry.

2 Algorytm backtrackingu

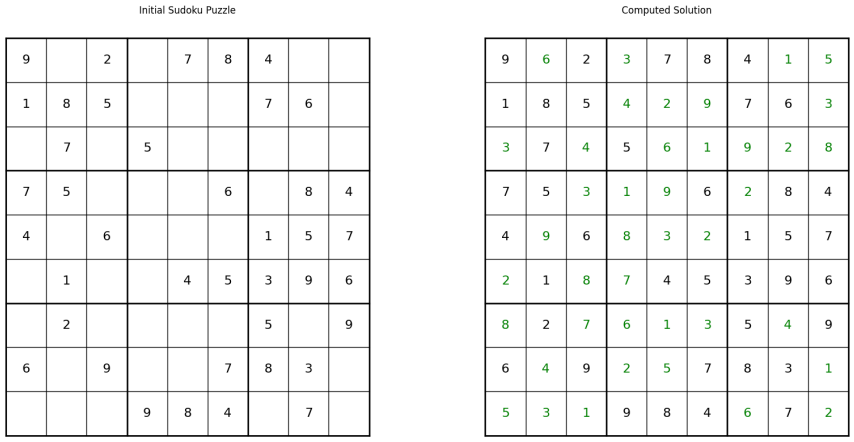
2.1 Zasada działania oraz wyniki

Algorytm rozwiązywania sudoku za pomocą backtrackingu jest klasycznym przykładem zastosowania rekurencji w rozwiązywaniu problemów kombinatorycznych.

```
def is_valid(board, row, col, num):
    for i in range(9):
        if board[row][i] == num or board[i][col] == num:
            return False

    start_row, start_col = 3 * (row // 3), 3 * (col // 3)
    for i in range(3):
        for j in range(3):
            if board[start_row + i][start_col + j] == num:
                return False

    return True

def solve_sudoku(board):
    empty = find_empty(board)
    if not empty:
        return True

    row, col = empty
    for num in range(1, 10):
        if is_valid(board, row, col, num):
            board[row][col] = num

            if solve_sudoku(board):
                return True

            board[row][col] = 0

    return False

def find_empty(board):
    for i in range(9):
        for j in range(9):
            if board[i][j] == 0:
                return (i, j)
    return None
```

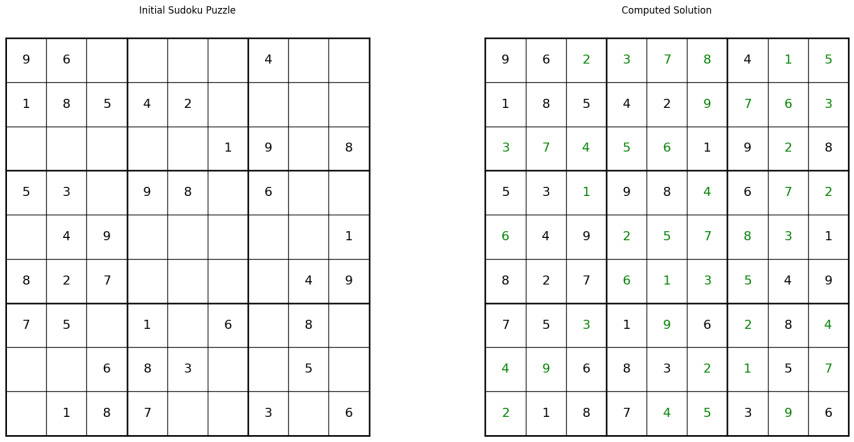
Algorytm szuka pierwszego pustego pola za pomocą find_empty. Wybiera pierwsze puste pole i próbuje wstawić kolejno liczby od 1 do 9. Sprawdzając każdą liczbę za pomocą funkcji is_valid, aby upewnić się, że nie łamie reguł Sudoku (czy liczba ta nie powtarza się w tej samej kolumnie, w tym samym rzędzie ani w tym samym 3x3 podkwadracie). Jeśli znajdzie liczbę, która pasuje, wstawia ją i rekurencyjnie próbuje rozwiązać resztę planszy. Jeśli rozwiązanie jest niemożliwe dla danej liczby, resetuje pole i wraca (backtracking) do poprzedniego kroku.

Wynik algorytmu dla prostego problemu (40/81 początkowo zapełnionych pól) daje 100% poprawnych odpowiedzi, co pokazane jest na
rysunku 1. Czas działania algorytmu jako średnia z 10 odpaleń wyniósł 0,020 s.



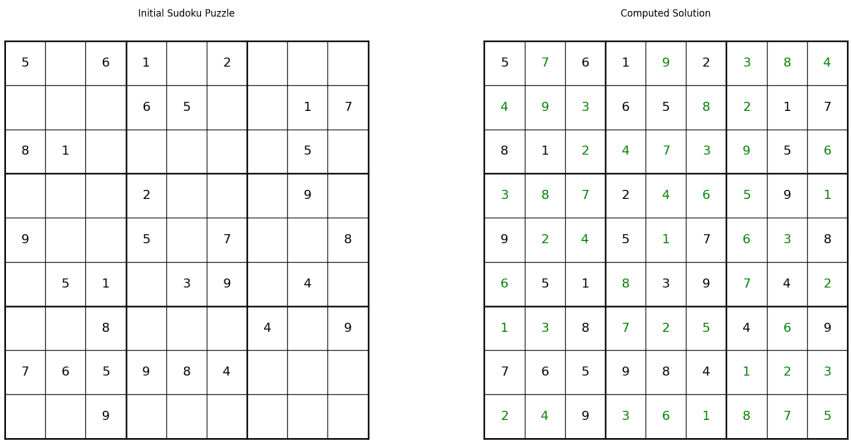
Rysunek 1: Wynik działania algorytmu brute-force dla prostego problemu.

Następnie sprawdzono działanie algorytmu na bardziej wymagającym problemie (38/81 początkowo zapełnionych pól). Wyniki widoczne są na rysunku 2. Czas działania algorytmu jako średnia z 10 odpaleń wyniósł 0,033 s.



Rysunek 2: Wynik działania algorytmu brute-force dla problemu średniej trudności.

Sprawdzono również działanie algorytmu na jeszcze bardziej wymagającym problemie (32/81 początkowo zapełnionych pól). Wyniki widoczne są na rysunku 3. Czas działania algorytmu jako średnia z 10 odpaleń wyniósł 0,041s.



5		6	1		2			
			6	5			1	7
8	1						5	
			2				9	
9			5		7			8
	5	1		3	9		4	
		8				4		9
7	6	5	9	8	4			
		9						

5	7	6	1	9	2	3	8	4
4	9	3	6	5	8	2	1	7
8	1	2	4	7	3	9	5	6
3	8	7	2	4	6	5	9	1
9	2	4	5	1	7	6	3	8
6	5	1	8	3	9	7	4	2
1	3	8	7	2	5	4	6	9
7	6	5	9	8	4	1	2	3
2	4	9	3	6	1	8	7	5

Rysunek 3: Wynik działania algorytmu brute-force dla problemu ciężkiego.

3 Algorytm genetyczny z wykorzystaniem biblioteki pygad

Zadaniem tego algorytmu genetycznego jest znalezienie jak najlepszego rozwiązania sudoku poprzez iteracyjne ulepszanie populacji rozwiązań przy użyciu operacji selekcji, krzyżowania i mutacji. Początkowa populacja to 50, liczba generacji to 10000. W implementacji algorytmu genetycznego do rozwiązywania sudoku użyto metody selekcji rodziców o nazwie "Steady-State Selection"(SSS). Metoda "Steady-State Selection"działa w następujący sposób:

- **Wybór Dwóch Najlepszych Osobników:** Najpierw z populacji wybiera się dwóch najlepszych osobników na podstawie wartości funkcji fitness. Ci osobnicy zostaną bezpośrednio przeniesieni do następnej generacji jako rodzice
- **Zastąpienie Najgorszych Osobników:** Następnie, w miejscu najgorszych osobników w populacji (osobniki z najniższą wartością funkcji fitness), umieszcza się nowych potomków wygenerowanych przez operacje krzyżowania i mutacji.

. Dlaczego Steady-State Selection?

- **Elitaryzm:** Metoda ta promuje elityzm poprzez zachowanie najlepszych osobników w każdej generacji. Dzięki temu zapewnia się, że najlepsze rozwiązania będą miały szansę przetrwania i ewoluować w kolejnych generacjach.
- **Zrównoważony Wzrost Populacji:** Populacja nie rośnie w sposób niekontrolowany, co jest szczególnie przydatne przy ograniczonych zasobach obliczeniowych lub pamięci.

3.1 Populacja początkowa

Populacja początkowa jest generowana przy użyciu funkcji create_initial_population(size). Każdy osobnik w populacji początkowej jest generowany poprzez wypełnienie pustych miejsc w siatce sudoku liczbami od 1 do 9, które nie powtarzają się w swoich wierszach, kolumnach i 3x3 podsiatkach. Takie podejście zapewnia dobry start, dzięki któremu wiele pozycji może być już poprawnie uzupełnionych.

```python
def create_initial_population(size):
    population = []
    for _ in range(size):
        individual = initial_sudoku.copy()
        for i in range(0, 9, 3):
            for j in range(0, 9, 3):
                subgrid = individual[i:i + 3, j:j + 3].flatten()
                missing_numbers = [num for num in range(1, 10)
                if num not in subgrid]
                np.random.shuffle(missing_numbers)
                subgrid[subgrid == 0] = missing_numbers
                individual[i:i + 3, j:j + 3] = subgrid.reshape((3, 3))
        population.append(individual.flatten())
    return np.array(population)


# Generate initial population
population_size = 50
initial_population = create_initial_population(population_size)
```

3.2 Krzyżowanie (Crossover)

Zaimplementowałem niestandardowe krzyżowanie (custom_crossover), które działa na zasadzie mieszania dwóch rodziców, zachowując części ich 3x3 podsiatek. Rodzice są losowo wybierani, a losowe 3x3 podsiatki od jednego rodzica są przekazywane do potomka.

```
def custom_crossover(parents, offspring_size, ga_instance):
    offspring = []
    for k in range(offspring_size[0]):
        parent1_idx = k % parents.shape[0]
        parent2_idx = (k + 1) % parents.shape[0]
        parent1 = parents[parent1_idx].reshape(9, 9)
        parent2 = parents[parent2_idx].reshape(9, 9)

        # Choose random grids from parent1 (mother)
        m = np.random.randint(1, 9)
        grids_ids = np.random.choice(range(9), m, replace=False)

        child = np.zeros((9, 9), dtype=int)
        for grid_id in range(9):
            i, j = divmod(grid_id, 3)
            if grid_id in grids_ids:
                child[i * 3:(i + 1) * 3, j * 3:(j + 1) * 3] =

                parent1[i * 3:(i + 1) * 3, j * 3:(j + 1) * 3]
            else:
                child[i * 3:(i + 1) * 3, j * 3:(j + 1) * 3] =

                parent2[i * 3:(i + 1) * 3, j * 3:(j + 1) * 3]

        # Ensure fixed positions are maintained
        child[fixed_positions] = initial_sudoku[fixed_positions]

        offspring.append(child.flatten())

    return np.array(offspring)
```

3.3 Funkcja fitness

Funkcja fitness (fitness_func) ocenia jakość każdego rozwiązania sudoku w populacji. Jest ona zdefiniowana w ten sposób, że im lepsze rozwiązanie, tym wyższa wartość fitness. Oto jak jest obliczana:

- **Powtarzające się liczby w wierszach i kolumnach**: Każda powtórzenie liczby w wierszu lub kolumnie jest penalizowane.
- **Unikalność liczb w 3x3 podsiatkach:** Każda unikalna liczba w każdej z 3x3 podsiatek jest premiowana.

Suma punktów odzwierciedla jakość rozwiązania - maksymalna liczba punktów wynosi 81 (jeśli każda liczba od 1 do 9 pojawia się dokładnie raz w każdym wierszu, kolumnie i 3x3 podsiatce).

```python
def fitness_func(ga_instance, solution, solution_idx):
    solution = solution.reshape((9, 9))
    score = 0

    # Penalize repeated numbers in rows and columns
    for i in range(9):
        row_counts = np.bincount(solution[i, :].astype(int))
        col_counts = np.bincount(solution[:, i].astype(int))
        score -= (np.count_nonzero(row_counts > 1) +
        np.count_nonzero(col_counts > 1))

    # Check 3x3 subgrids
    for i in range(0, 9, 3):
        for j in range(0, 9, 3):
            subgrid = solution[i:i + 3, j:j + 3].flatten()
            score += len(set(subgrid))

    return score
```

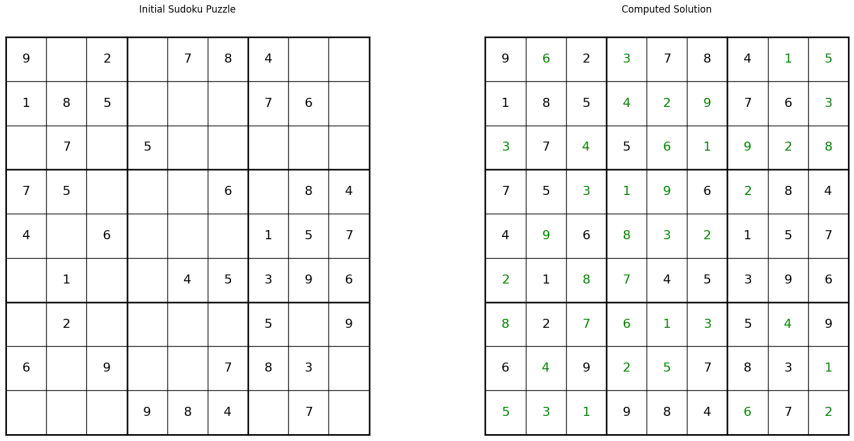
3.4 Mutacje

Mutacja (custom_mutation) polega na losowej zamianie dwóch różnych pozycji w ramach losowo wybranej 3x3 podsiatki. Ta operacja pomaga wprowadzać różnorodność genetyczną w populacji, co może przyspieszyć osiągnięcie lepszego rozwiązania.

```python
def custom_mutation(offspring, ga_instance):
    for idx in range(offspring.shape[0]):
        individual = offspring[idx].reshape((9, 9))
        # Choose a random 3x3 subgrid
        grid_id = np.random.randint(9)
        i, j = divmod(grid_id, 3)
        subgrid = individual[i * 3:(i + 1) * 3, j * 3:(j + 1) * 3].flatten()

        # Swap two random positions within the subgrid
        pos1, pos2 = np.random.choice(9, 2, replace=False)
        subgrid[pos1], subgrid[pos2] = subgrid[pos2], subgrid[pos1]

        # Ensure fixed positions are maintained
        individual[i * 3:(i + 1) * 3, j * 3:(j + 1) * 3] = subgrid.reshape((3, 3))
        individual[fixed_positions] = initial_sudoku[fixed_positions]

        offspring[idx] = individual.flatten()
    return offspring
```

3.5 Wyniki

Wynik algorytmu dla prostego problemu (40/81 początkowo zapełnionych pól) daje 100% poprawnych odpowiedzi, co pokazane jest na rysunku 4.
Czas działania algorytmu jako średnia z 10 odpaleń wyniósł 12,320 s.

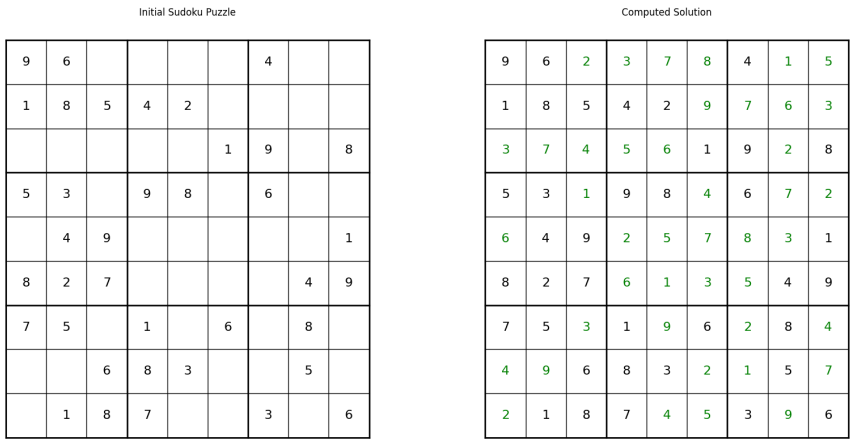


9		2		7	8	4		
1	8	5				7	6	
	7		5					
7	5				6		8	4
4		6				1	5	7
	1			4	5	3	9	6
	2					5		9
6		9			7	8	3	
			9	8	4		7	

9	6	2	3	7	8	4	1	5
1	8	5	4	2	9	7	6	3
3	7	4	5	6	1	9	2	8
7	5	3	1	9	6	2	8	4
4	9	6	8	3	2	1	5	7
2	1	8	7	4	5	3	9	6
8	2	7	6	1	3	5	4	9
6	4	9	2	5	7	8	3	1
5	3	1	9	8	4	6	7	2

Rysunek 4: Wynik działania algorytmu genetycznego dla prostego problemu.

Następnie sprawdzono działanie algorytmu na bardziej wymagającym problemie (38/81 początkowo zapełnionych pól). Wyniki widoczne są na rysunku 5. Czas działania algorytmu jako średnia z 10 odpaleń wyniósł 32,041 s.

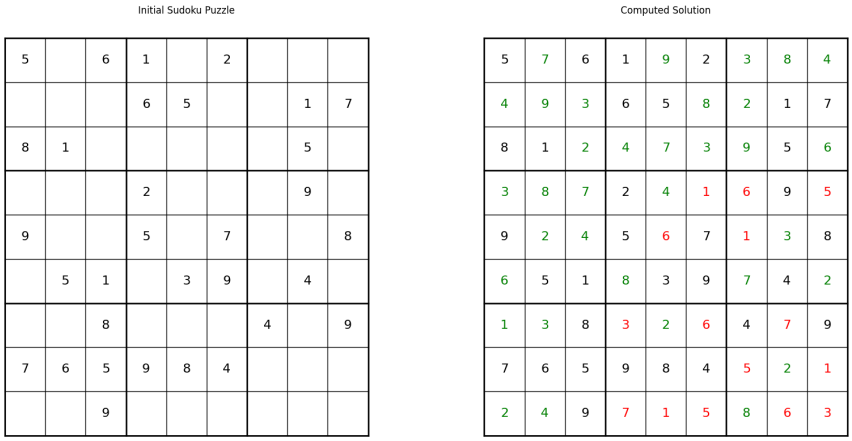


9	6					4		
1	8	5	4	2				
					1	9		8
5	3		9	8		6		
	4	9						1
8	2	7					4	9
7	5		1		6		8	
		6	8	3			5	
	1	8	7			3		6

9	6	2	3	7	8	4	1	5
1	8	5	4	2	9	7	6	3
3	7	4	5	6	1	9	2	8
5	3	1	9	8	4	6	7	2
6	4	9	2	5	7	8	3	1
8	2	7	6	1	3	5	4	9
7	5	3	1	9	6	2	8	4
4	9	6	8	3	2	1	5	7
2	1	8	7	4	5	3	9	6

Rysunek 5: Wynik działania algorytmu genetycznego dla problemu średniej trudności.

Sprawdzono również działanie algorytmu na jeszcze bardziej wymagającym problemie (32/81 początkowo zapełnionych pól). Wyniki widoczne są na rysunku 6. Czas działania algorytmu jako średnia z 10 odpaleń wyniósł 134,650s. W przypadku tak zaawansowanego problemu algorytm często zatrzymuje się w lokalnym minimum i nie potrafi się już poprawić, aż do końca swojego działania. Dla podanego problemu algorytm osiąga dokładność poprawnych odpowiedzi na poziomie około 70%.



Rysunek 6: Wynik działania algorytmu genetycznego dla problemu ciężkiego.

4 Algorytm ACO

Algorytm ACO (Ant Colony Optimization) to metaheurystyka inspirowana zachowaniem rzeczywistych kolonii mrówek, używana głównie do rozwiązywania problemów optymalizacji kombinatorycznej. **Jak działa ACO?**

- **Inicjalizacja:** Na początku procesu wszystkie krawędzie (ścieżki) mają przypisane początkowe wartości feromonów.
- **Budowanie rozwiązań:** Każda mrówka konstruuje swoje rozwiązanie, wybierając kolejne kroki na podstawie feromonów i heurystyki. Wartości feromonów są aktualizowane w trakcie tego procesu.
- **Aktualizacja feromonów:** Po zakończeniu konstrukcji rozwiązań przez wszystkie mrówki, feromony na ścieżkach są aktualizowane: dodaje się feromony na ścieżkach, które były częścią dobrych rozwiązań, a jednocześnie feromony na wszystkich ścieżkach odparowują w określonym tempie.
- **Iteracja:** Proces powtarza się przez ustaloną liczbę iteracji lub do spełnienia warunku stopu (np. znalezienia satysfakcjonującego rozwiązania).

4.1 Rozwiązanie przy pomocy biblioteki aco

```
class SudokuAntColony(AntColony):
    def __init__(self, problem, ant_count=50, iterations=100, alpha=1.0,
    beta=1.0, rho=0.5, Q=1.0):
        self.problem = problem
        self.nodes = [(i, j) for i in range(9) for j in range(9) if problem[i, j] == 0]
        self.pheromone = np.ones((9, 9, 9))
        self.alpha = alpha
        self.beta = beta
        self.rho = rho
        self.Q = Q
        super().__init__(self.nodes, ant_count=ant_count, iterations=iterations,
        alpha=alpha, beta=beta, pheromone_evaporation_rate=rho, pheromone_constant=Q)
```

- **problem:** Przechowuje początkowy problem (nierozwiązane Sudoku).
- **nodes** Lista współrzędnych pustych pól (węzłów) w Sudoku.
- **pheromone:** Trójwymiarowa macierz feromonów. Każdy element [i, j, k] reprezentuje ilość feromonu dla liczby k+1 na pozycji (i, j).
- **alpha:** Wpływ feromonów na wybór trasy.
- **beta:** Wpływ heurystyki na wybór trasy
- **rho:** Współczynnik parowania feromonów.
- **Q:** Stała feromonowa.

```
def _update_pheromone(self, ant):
    solution_matrix = self.problem.copy()
    for (i, j), num in zip(self.nodes, ant.path):
        solution_matrix[i, j] = num + 1
    score = evaluate(solution_matrix)
    for (i, j), num in zip(self.nodes, ant.path):
        self.pheromone[i, j, num] += self.Q / score
```

Solution_matrix to kopia problemu Sudoku, do której dodawane są ruchy mrówki. Evaluate: Ocena jakości rozwiązania (wyższy wynik oznacza lepsze rozwiązanie). Ilość feromonów jest zwiększana proporcjonalnie do jakości rozwiązania.

```
def _get_probabilities(self, ant, index):
    (i, j) = self.nodes[index]
    valid_numbers = [num for num in range(9) if self.is_valid_move(ant, i, j, num + 1)]
    pheromones = np.array([self.pheromone[i, j, num] for num in valid_numbers])
    heuristics = np.ones(len(valid_numbers))
    probabilities = (pheromones ** self.alpha) * (heuristics ** self.beta)
    return probabilities / probabilities.sum(), valid_numbers
```

- **valid_numbers:** Lista liczb (0-8) dozwolonych na pozycji (i, j) (sprawdzana przez is_valid_move).
- **pheromones:** Macierz feromonów dla dozwolonych liczb.
- **heuristics:** Macierz heurystyk (wszystkie wartości ustawione na 1, co oznacza, że heurystyka nie wpływa na wybór).
- **probabilities:** Prawdopodobieństwa wyboru każdej dozwolonej liczby na podstawie feromonów i heurystyk.

```
def get_path(self):
    best_path = None
    best_score = -1
    for _ in range(self.iterations):
        self.iteration()
        for ant in self.ants:
            solution_matrix = self.problem.copy()
            for (i, j), num in zip(self.nodes, ant.path):
                solution_matrix[i, j] = num + 1
            score = evaluate(solution_matrix)
            if score > best_score:
                best_score = score
                best_path = ant.path
    return best_path
```

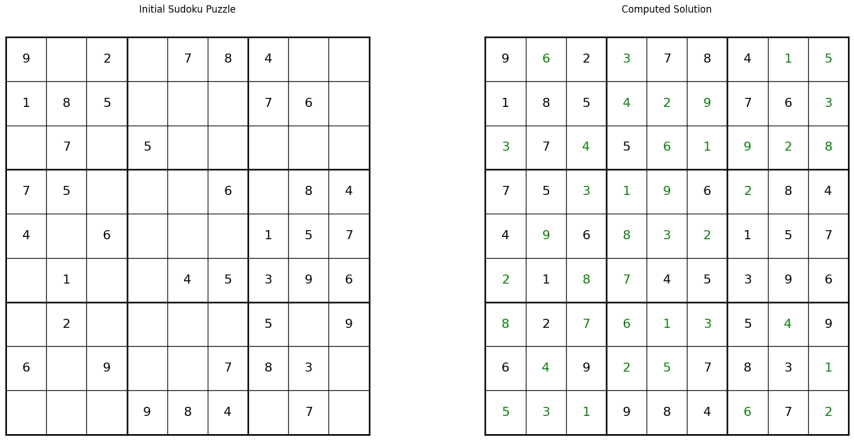
W każdej iteracji mrówki przeszukują przestrzeń rozwiązań. Oceniane są rozwiązania i wybierane najlepsze znalezione do tej pory.

```
 def iteration(self):
    self.ants = [self.create_ant() for _ in range(self.ant_count)]
    for ant in self.ants:
        for index in range(len(self.nodes)):
            probabilities, valid_numbers = self._get_probabilities(ant, index)
            if len(valid_numbers) == 0:
                break
            chosen_number = np.random.choice(valid_numbers, p=probabilities)
            ant.path.append(chosen_number)
        self._update_pheromone(ant)
```

Tworzy nowe mrówki.Każda mrówka wybiera liczby dla pustych pól Sudoku na podstawie prawdopodobieństw obliczonych z feromonów i heurystyk. Po zakończeniu trasy przez mrówkę, aktualizowane są feromony.

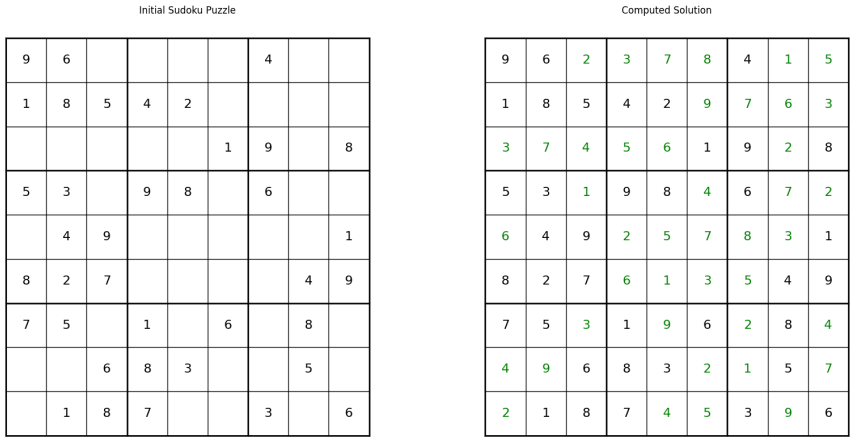
4.2 Wyniki dla 150 iteracji oraz 50 mrówek

Wynik algorytmu dla prostego problemu (40/81 początkowo zapełnionych pól) daje 100% poprawnych odpowiedzi, co pokazane jest na rysunku 7.
Czas działania algorytmu jako średnia z 10 odpaleń wyniósł 29,412 s.



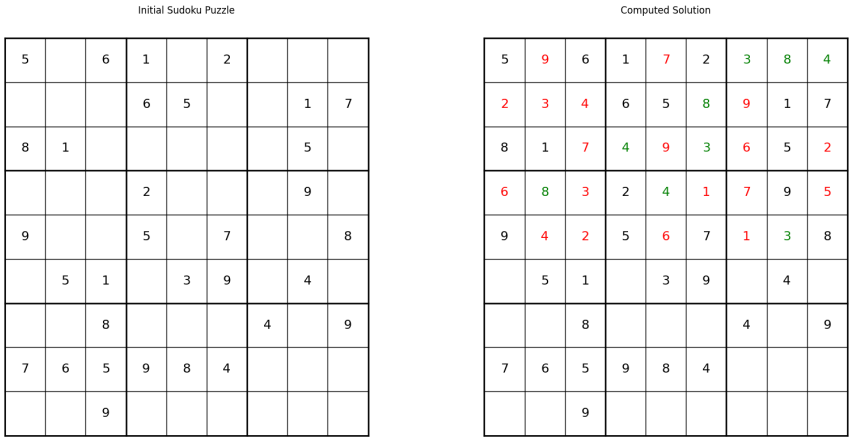
Rysunek 7: Wynik działania algorytmu ACO dla prostego problemu.

Następnie sprawdzono działanie algorytmu na bardziej wymagającym problemie (38/81 początkowo zapełnionych pól). Wyniki widoczne są na rysunku 8. Czas działania algorytmu jako średnia z 10 odpaleń wyniósł 34,621 s.



Rysunek 8: Wynik działania algorytmu ACO dla problemu średniej trudności.

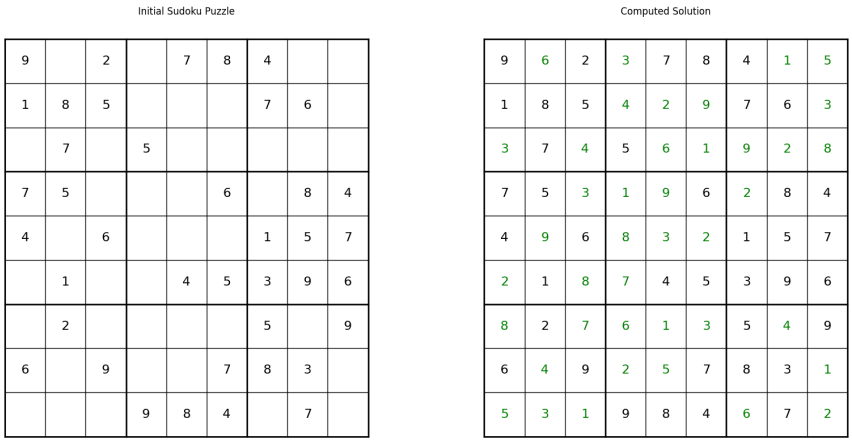
Sprawdzono również działanie algorytmu na jeszcze bardziej wymagającym problemie (32/81 początkowo zapełnionych pól). Wyniki widoczne są na rysunku 9. Czas działania algorytm jako średnia z 10 odpaleń wyniósł 45,650s. W przypadku tak zaawansowanego problemu algorytm osiąga bardzo słabe wyniki. Dla podanego problemu algorytm osiąga dokładność poprawnych odpowiedzi na poziomie około 20%.



Rysunek 9: Wynik działania algorytmu ACO dla problemu ciężkiego.

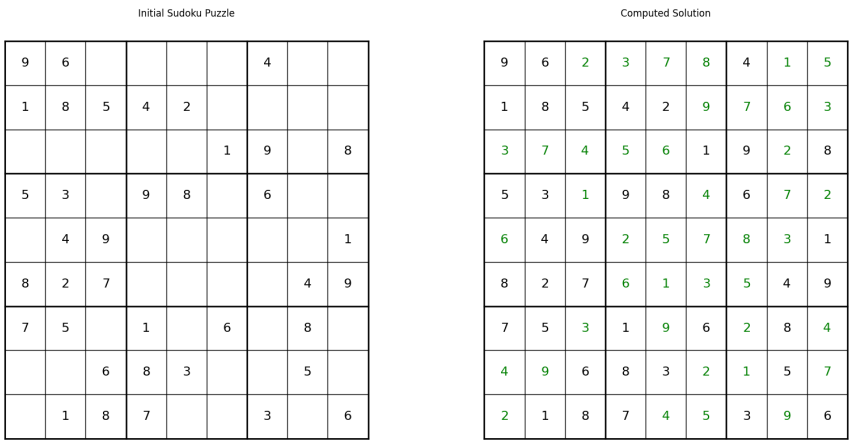
4.3 Wyniki dla 300 iteracji oraz 75 mrówek

Wynik algorytmu dla prostego problemu (40/81 początkowo zapełnionych pól) daje 100% poprawnych odpowiedzi, co pokazane jest na rysunku 10.
Czas działania algorytmu jako średnia z 10 odpaleń wyniósł 190,573 s.



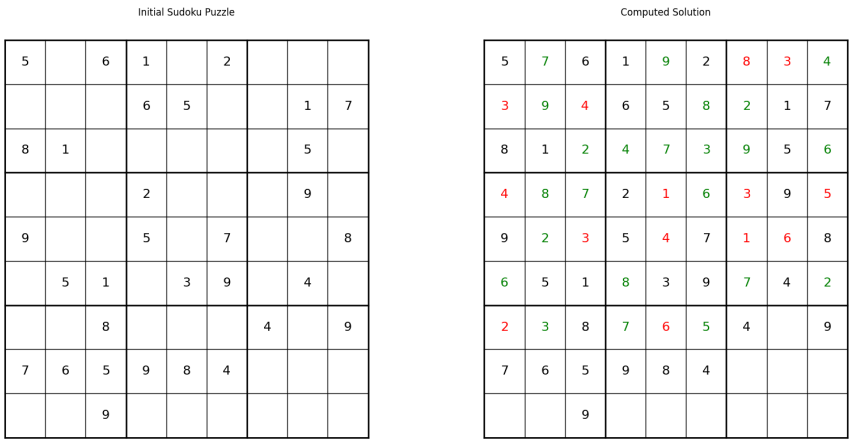
Rysunek 10: Wynik działania algorytmu ACO z 300 iteracjami dla prostego problemu.

Następnie sprawdzono działanie algorytmu na problemie średniej trudności (38/81 początkowo zapełnionych pól). Wyniki widoczne są na rysunku 11. Czas działania algorytmu jako średnia z 10 odpaleń wyniósł 275,180 s.



Rysunek 11: Wynik działania algorytmu ACO z 300 iteracjami dla problemu średniej trudności.

Sprawdzono również działanie algorytmu na jeszcze bardziej wymagającym problemie (32/81 początkowo zapełnionych pól). Wyniki widoczne są na rysunku 12. Czas działania algorytmu jako średnia z 10 odpaleń wyniósł 359,253s. Po zwiększeniu liczby iteracji oraz dodaniu mrówek wynik znacząco się polepszył, jednak algorytm wykonuje się znacznie dłużej. Dla podanego problemu algorytm osiąga dokładność poprawnych odpowiedzi na poziomie około 53%.



Rysunek 12: Wynik działania algorytmu ACO z 300 iteracjami dla problemu ciężkiego.

5 Podsumowanie

Podsumowanie całej pracy można zaobserbować w tabeli 1.

Tabela 1: Porównanie wyników różnych algorytmów dla różnych poziomów trudności Sudoku

	Algorytm Backtracking	Algorytm Genetyczny	ACO (100 iteracji 50 mrówek)	ACO (300 iteracji 75 mrówek)
Problem prosty	100%(0,020s)	100%(12,320s)	100%(29,412s)	100%(190,573s)
Problem średniej trudności	100%(0,033s)	100%(32,041s)	100%(34,621)	100%(275,180s)
Problem trudny	100%(0,041s)	70%(134,650s)	20%(45,650s)	53%(359,253s)

- **Algorytm Backtracking**
  - Jest bardzo skuteczny dla wszystkich testowanych problemów, gdzie osiąga 100% poprawnych rozwiązań.
  - Czas wykonania rośnie wraz z trudnością problemu, ale pozostaje na stosunkowo niskim poziomie, nawet dla problemu trudnego.
- **Algorytm Genetyczny**
  - Wykazuje wysoką skuteczność dla problemów prostych i średniej trudności, osiągając 100% poprawnych rozwiązań.
  - Czas wykonania znacznie rośnie wraz z trudnością problemu. Dla problemu trudnego czas wykonania jest znacząco wyższy w porównaniu do algorytmu Backtracking.
- **ACO (Ant Colony Optimization) - 100 iteracji, 50 mrówek**
  - Dla problemu prostego i problemu średniej trudności osiąga 100% poprawnych rozwiązań, z problemem trudnym radzi sobie bardzo słabo.
  - Czas wykonania jest znacznie dłuższy niż w przypadku algorytmów deterministycznych, ale nadal akceptowalny dla problemów prostych i średniej trudności.
- **ACO (Ant Colony Optimization) - 300 iteracji, 75 mrówek**
  - Dla problemu trudnego osiąga zdecydowanie lepsze wyniki skuteczności (około 30 punktów procentowych więcej) niż inna wersja tego algorytmu, co sugeruje zależność od liczby iteracji i mrówek.
  - Czas wykonania jest najdłuższy spośród wszystkich testowanych algorytmów, co może być kosztem uzyskania lepszych wyników dla problemów bardziej złożonych.

Podsumowując, wybór odpowiedniego algorytmu do rozwiązywania Sudoku zależy od poziomu trudności problemu oraz wymagań co do czasu wykonania. Algorytmy deterministyczne jak Backtracking są szybkie i skuteczne dla problemu rozwiązywania Sudoku. Natomiast metaheurystyki takie jak Algorytm Genetyczny i ACO dobrze radzą sobie z problemami mniej złożonymi, jednak potrzebują dłuższego czasu obliczeń.