Dokumentacja projektu

Rozwiązywanie i weryfikacja Sudoku

Bartosz Bieniek, grupa 4.7

14 stycznia 2026

Spis treści

1 Wstęp

Celem projektu jest zaprojektowanie i implementacja aplikacji służącej do walidacji oraz rozwiązywania łamigłówek typu Sudoku. System uruchamiany jest z poziomu skryptu powłoki, który koordynuje przetwarzanie danych, a na podstawie uzyskanych wyników tworzy raport i jego kopię zapasową.

2 Definicja problemu

Sudoku to gra logiczna, która polega na uzupełnieniu planszy 9×9 cyframi od 1 do 9 w taki sposób, aby każda z nich występowała dokładnie raz w każdym wierszu, kolumnie i bloku 3×3. Część pól jest wstępnie wypełniona, co narzuca ograniczenia na pozostałe pozycje. Zadaniem algorytmu jest wyznaczenie wszystkich możliwych rozwiązań, przy czym dla dobrze zadanego problemu powinno istnieć dokładnie jedno rozwiązanie.

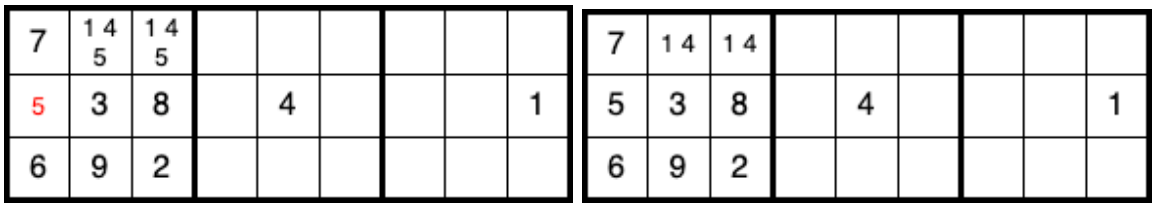
3 Rozwiązywanie Sudoku

3.1 Techniki deterministyczne

Techniki deterministyczne polegają na analizie aktualnie wpisanych na planszy wartości oraz list kandydatów. Ich celem jest odnajdowanie sytuacji, w których dana cyfra może zostać jednoznacznie wpisana, bez konieczności zgadywania.

Naked Singles

Polega na wpisywaniu wartości w pola, które posiadają tylko jedną dopuszczalną wartość na liście kandydatów. Dzieje się tak w sytuacji, gdy pozostałe pozostałe możliwości zostały wyeliminowane wskutek wpisywania cyfr w danym wierszu kolumnie lub bloku.



Rysunek 1: Przykład zastosowania techniki Naked Singles.

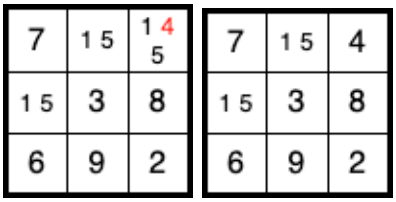
Listing 1: Przykładowa implementacja wyszukiwania Naked Singles w języku Python.

```python
def find_naked_singles(board: Board) -> list[tuple[int, Position]]:
    naked_singles: list[tuple[int, Position]] = []

    for position in np.ndindex((9, 9)):
        candidates = board.get_candidates(position)

        if len(candidates) == 1:
            naked_singles.append((list(candidates)[0], position))
```

```
9
10      return naked_singles
```

Hidden Singles

Polega na analizie wierszy, kolumn i bloków w poszukiwaniu wartości, które mogą wystąpić tylko w jednej z należących do nich komórek. Fakt, że dana cyfra nie może pojawić się w innym miejscu sprawdzanego obszaru, wymusza jej wpisanie.



Rysunek 2: Przykład zastosowania techniki Hidden Singles.

Listing 2: Przykładowa implementacja wyszukiwania Hidden Singles w języku Python.

```python
def find_hidden_single(board: Board, positions: list[Position])\
        -> list[tuple[int, Position]]:
    counts = [0] * 10
    history: list[Position | None] = [None] * 10

    for position in positions:
        if board.get_value(position) != 0:
            continue
        for candidate in board.get_candidates(position):
            counts[candidate] += 1
            history[candidate] = position

    hidden_singles: list[tuple[int, Position]] = []

    for value in range(1, 10):
        if counts[value] != 1:
            continue
        position = history[value]
        hidden_singles.append((value, position))

    return hidden_singles
```

3.2 Przeszukiwanie z nawrotami (backtracking)

W sytuacji gdy techniki deterministyczne nie prowadzą do dalszych postępów, algorytm przechodzi do przeszukiwania z nawrotami. Dla jednej z niewypełnionych komórek wybierany jest jeden z dostępnych kandydatów i traktowany jako hipoteza. Plansza z wpisaną wartością jest ponownie poddawana rozwiązywaniu technikami deterministycznymi i przeszukiwania z nawrotami. Jeżeli przyjęta wartość doprowadzi do rozwiązania, algorytm cofa się i próbuje innej

możliwości. Proces ten jest powtarzany aż do momentu znalezienia wszystkich poprawnych rozwiązań.

Do wyboru pola wykorzystywana jest heurystyka minimalnej ilości kandydatów (*Minimum Remaining Values*), polegająca na wyszukaniu komórki z najmniejszą liczbą dozwolonych wartości. Zmniejsza to liczbę możliwych odgałęzień i przyspiesza wyszukiwanie rozwiązania.

Listing 3: Przykładowa implementacja algorytmu przeszukiwania z nawrotami w języku Python.

```python
def solve(board: Board) -> list[Board]:
    while True:
        naked_singles = find_naked_singles(board)

        for value, position in naked_singles:
            if value not in board.get_candidates(position):
                return []
            board.place(value, position)

        hidden_singles = find_hidden_singles(board)

        for value, position in hidden_singles:
            if value not in board.get_candidates(position):
                return []
            board.place(value, position)

        if len(naked_singles) == 0 and len(hidden_singles) == 0:
            break

    if board.is_solved():
        return [board]

    solutions = []
    pivot_position = mrv_find_pivot_position(board)

    for candidate in board.get_candidates(pivot_position):
        assumed_board = copy.deepcopy(board)
        assumed_board.place(candidate, pivot_position)
        solutions.extend(solve(assumed_board))

    return solutions
```

4 Weryfikacja poprawności rozwiązań

Weryfikacja rozwiązań Sudoku polega na sprawdzeniu, czy wypełniona plansza spełnia wszystkie reguły gry, a więc czy w każdym wierszu, kolumnie oraz bloku 3×3 każda z cyfr występuje dokładnie raz. Oznacza to jednocześnie, że w rozwiązaniu nie występują puste pola.

Sprawdzenie to może zostać zrealizowane na dwa sposoby. Pierwszy z nich polega na bezpośrednim wykrywaniu duplikatów w poszczególnych obszarach planszy. Drugi natomiast wykorzystuje listy kandydatów, wykreślając dopuszczalne wartości w miarę wpisywania kolejnych cyfr. Rozwiązanie jest poprawne, gdy każda wstawiana liczba była w danym momencie dozwolona na swojej pozycji.

4.1 Sprawdzanie duplikatów

Pierwszy wariant algorytmu sprawdza dla każdego wiersza, kolumny i bloku 3×3, czy zawiera dokładnie jedno powtórzenie każdej cyfry oraz że żadna komórka nie jest pusta.

Data: Tablica liczb w obszarze w
Result: *true* lub $false$
for $i \leftarrow 1..9$ **do**
  if $w[i] = 0$ **then**
    Wypisz $false$ i zakończ działanie programu.
  end
  for $j \leftarrow 1..9$ **do**
    if $i \neq j$ *oraz* $w[i] = w[j]$ **then**
      Wypisz $false$ i zakończ działanie programu.
    end
  end
end
Wypisz *true* i zakończ działanie programu.

Algorithm 1: Przykładowa implementacja weryfikacji przez duplikaty.

4.2 Listy kandydatów

Drugi wariant algorytmu wykorzystuje listy kandydatów, w którym dla każdej komórki przechowywany jest zbiór dopuszczalnych wartości, początkowo zawierający wszystkie liczby $1, 2, \ldots, 9$. W miarę uzupełniania pomocniczej planszy kolejnymi liczbami z rozwiązania, program usuwa wartości z list kandydatów w odpowiadającym ich wierszach, kolumnach oraz blokach 3×3. Jeżeli w dowolnym momencie dana cyfra nie może zostać wpisana na swojej pozycji, rozwiązanie uznawane jest za niepoprawne.

Data: Rozwiązanie s (tablica 9×9)
Result: *true* lub $false$
$p \leftarrow$ tablica 9×9 wypełniona zbiorami $(1, 2, \ldots, 9)$
for $i \leftarrow 1..9$ **do**
  for $j \leftarrow 1..9$ **do**
    if $p[i][j]$ *nie zawiera* $s[i][j]$ **then**
      Wypisz $false$ i zakończ działanie programu.
    end
    Usuń wartość $s[i][j]$ z listy kandydatów p dla wszyskich pól w i-tym wierszu, j-tej kolumnie oraz bloku 3×3.
  end
end
Wypisz *true* i zakończ działanie programu.

Algorithm 2: Przykładowa implementacja weryfikacji wykorzystującej listy kandydatów.

5 Implementacja

5.1 Struktura katalogów

- `./main.sh` - główny skrypt uruchamiający całe rozwiązanie.
- `./fetch_puzzles.sh` - skrypt pobierający z internetu nierozwiązane plansze.
- `./data/input` - katalog z danymi wejściowymi.
- `./data/output` - katalog z danymi wyjściowymi.
- `./data/reports` - katalog z wygenerowanymi raportami.
- `./data/backups` - katalog z kopiami zapasowymi raportów.
- `./templates/report_template.j2` - szablon raportu.
- `./src/solver/board.py` - plik zawierający model planszy Sudoku wraz z najważniejszymi operacjami, jakie można na niej przeprowadzać.
- `./src/solver/solver.py` - główny skrypt odpowiedzialny za rozwiązywanie planszy.
- `./src/solver/validator.py` - skrypt odpowiedzialny za sprawdzenie poprawności rozwiązania.
- `./src/backup_reports.py` - skrypt tworzący kopię zapasową raportów.
- `./src/create_report.py` - skrypt generujący raport na podstawie danych wyjściowych.
- `./src/solve_puzzle.py` - skrypt przetwarzający dane wejściowe, uruchamiający rozwiązywanie łamigłówki i mierzący czas.

5.2 Instrukcja obsługi

W celu uruchomienia programu należy przejść do głównego folderu projektu, a następnie z wiersza poleceń uruchomić skrypt `main.sh`.

Listing 4: Uruchomienie.

```
cd ~/Downloads/sudoku/
./main.sh
```

Aplikacja automatycznie rozpocznie czytanie wejściowych plansz z folderu `./data/input/`, a ich rozwiązania zapisze w katalogu `./data/output/` w plikach o identycznych nazwach.

Pliki wejściowe składają się z pojedynczego, osiemdziesięciojedno znakowego ciągu cyfr, będącego reprezentacją planszy Sudoku czytanej wierszami. Liczby od 1 do 9 włącznie odpowiadają wpisanym cyfrom, a 0 pustym kratkom.

Listing 5: Przykładowe dane wejściowe.

```
013508000790300802006004003000100000100035070000060480070009004000050068061080905
```

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

Rysunek 3: Wizualizacja planszy wejściowej.

Pliki wyjściowe składają się z przynajmniej dwóch linii, w których pierwsza zawiera informacje o planszy wejściowej, ilości rozwiązań oraz czasie rozwiązywania wyrażonym w sekundach. Kolejne wiersze odpowiadają natomiast otrzymanym rozwiązaniom.

Listing 6: Przykładowe dane wyjściowe.

```
1 013508000790300802006004003000100000100035070000060480070009004000050068061080905,2,0.002534

2 213598647794316852586274193628147539149835276357962481875629314932451768461783925
3 213598647794316852856274193687142539149835276325967481578629314932451768461783925
```

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

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

Rysunek 4: Wizualizacja wygenerowanych rozwiązań.

Na podstawie danych zawartych w folderze `./data/output` program przygotuje i otworzy w przeglądarce raport dotyczący otrzymanych wyników. Dodatkowo, w katalogu `./data/backups` zostanie utworzona kopia zapasowa wszystkich raportów, które z różnych względów jeszcze jej nie posiadają.

5.3 Instrukcja wdrożenia

Do uruchomienia programu wymagany jest komputer działający pod kontrolą systemu macOS lub Linux z dostępem do internetu (do instalacji wymaganych pakietów) oraz interpreter

języka Python w wersji 3.14.

Pobrane archiwum należy rozpakować, a następnie przejść do głównego katalogu projektu.

Listing 7: Rozpakowanie projektu.

```
cd ~/Downloads/
unzip bbieniek_sudoku_projekt.zip -d ~/Downloads/sudoku
cd ~/Downloads/sudoku/
```

Do działania programu konieczne jest również instalacja wykorzystywanych pakietów.

Listing 8: Instalacja wymaganych pakietów.

```
cd ~/Downloads/sudoku/
pip install -r requirements.txt
```

Dane wejściowe mogą zostać utworzone ręcznie w podanym wcześniej formacie lub pobrane z internetu w formie gotowych plansz, przy użyciu skryptu `fetch_puzzles.sh`.

Listing 9: Pobranie z internetu plansz do rozwiązania.

```
cd ~/Downloads/sudoku/
./fetch_puzzles.sh ./data/input/
```

Dodatkowe informacje

Rozwiązanie zostało przygotowane i sprawdzone w systemie o poniższej specyfikacji.

- Apple MacBook Pro, M1 Pro (ARMv8.6-A), 16 GB RAM
- System operacyjny macOS 26.2 (25C56)
- Interpreter Python 3.14