Foundations of artificial intelligence
Sudoku Solver

Davide Pizzolato - 881347

November 2022

1 Introduction

This document is intended to provide a description of how I solved the assignment 1.

2 Sudoku base class

The ***Sudoku*** class is a base class that decode the string representation of a sudoku and provide some basic method like *set()*, *get()* and *is_correct()*. It also define the abstract method *solve()*.

3 Solver based on constraint propagation and backtracking

The class ***SudokuConstraints*** is derived from ***Sudoku*** and override the *set()*.

In this class I keep a list of constraints for each cell, this list is updated when a cell is set (also during initialization). Each cell's constraints list contains 9 integer that represent the amount of digit i contained in the same row, column or box.

Listing 1: Overrided set function

```
def set(self, r: int, c: int, val: Optional[int]):
    # Nothing change
    if self.data[r][c] == val:
        return

    # Unset old constraints
    if self.data[r][c] is not None:
        self.__set_constraint(r, c, self.data[r][c], False)
    # Set new constraints
    if val is not None:
        self.__set_constraint(r, c, val, True)

    super().set(r, c, val)
```

_get_possible_values_for() is easily defined (with complexity $O(1)$) as the digits that have value 0 in the constraints list.

Listing 2: _get_possible_values_for function

```
def _get_possible_values_for(self, r: int, c: int) -> Set[int]:
    out = set()
    for i in range(9):
        if self.__constraints[r][c][i] == 0:
            out.add(i + 1)
    return out
```

I implement the *solve()* function with a simple backtracking that use *_get_possible_values_for()* to cut out some branch.

Listing 3: solve function

```
def solve(self) -> bool:
```

```python
    def func(r: int, c: int) -> bool:
        if r == 9:
            return True # Reach bottom of recursion

        next_r, next_c = self.inc_row_col(r, c)
        if self.get(r, c) is not None:
            # Fixed number
            return func(next_r, next_c)
        else:
            choices = self._get_possible_values_for(r, c)
            for i in choices:
                self.set(r, c, i)
                if func(next_r, next_c):
                    return True
            self.set(r, c, None)

        return False # Wrong solution
    return func(0, 0)
```

4 Solver based on constraint propagation, backtracking and greedy ordering

This class is called ***SudokuConstraintsGreedy*** and inherits from ***SudokuConstraints***.

The last solution works but it is a bit naive, I rewrite it in order to always set the cell that has the minimum possible values (ideally $1 \mapsto$ the choice is forced, or $0 \mapsto$ wrong solution).

The overridden *solve()* function is the following:

Listing 4: solve function

```python
def solve(self) -> bool:
    next_step = self.__get_next_step()
    if next_step[0] == self.__NextStep.FoundZero:
        return False # Wrong solution
    if next_step[0] == self.__NextStep.NotFound:
        return True # Reach bottom of recursion

    r, c = next_step[1]

    choices = self._get_possible_values_for(r, c)
    for i in choices:
        self.set(r, c, i)
        if self.solve():
            return True
    self.set(r, c, None)

    return False # Wrong solution
```

5 Solver with genetic algorithm

The last solver uses the genetic approach, the class is ***SudokuGenetic*** and it inherit from ***Sudoku***.

During the creation of the first generation the indirect constrain that each digit must appear in each row is respected, and during all operation this constraints is preserved.

For example the *mutate()* function swap cell on the same row.

5.1 Solve method

In the solve function i create the initial generation and then I iterate through generation. On every iteration I preserve the first ELITE_N items and then I generate the remaining child. The child are generated with crossover and then mutated with a probability of *__mutation_rate*. The mutation rate are adjusted with Rechenberg's 1/5 rule.

Listing 5: solve function

```
def solve(self):
    gen = self.__get_first_gen(GEN_N)
    gen.sort(key=lambda x: x.__rank_solution())

    gen_n = 0
    while gen[0].__rank_solution() != 0:
        cur_rank = gen[0].__rank_solution()

        gen_n += 1
        print("Fitness: " + str(cur_rank) + ", Generation: " + str(gen_n))
        print("\n")

        # keep elite
        new_gen = gen[0:ELITE_N]
        candidates = gen[0:CANDIDATE_N]

        # generate others
        num_of_mutations = 0
        successful_mutations = 0
        while len(new_gen) < GEN_N:
            p1 = self.__get_parent(candidates)
            p2 = self.__get_parent(candidates)
            c1, c2 = p1.__generate(p2)

            # Mutate
            c1_old_rank = c1.__rank_solution()
            c2_old_rank = c2.__rank_solution()
            if c1.__mutate():
                num_of_mutations += 1
                if c1.__rank_solution(recalculate=True) < c1_old_rank:
                    successful_mutations += 1
            if c2.__mutate():
                num_of_mutations += 1
                if c2.__rank_solution(recalculate=True) < c2_old_rank:
                    successful_mutations += 1

            # Append
            new_gen.append(c1)
            new_gen.append(c2)

        gen = new_gen
        gen.sort(key=lambda x: x.__rank_solution())

        # Rechenberg's 1/5 rule
        if num_of_mutations != 0:
            rate_of_successful_mutations = successful_mutations / num_of_mutations
            if rate_of_successful_mutations < (1/5):
                self.__mutation_rate *= 0.95
            elif rate_of_successful_mutations > (1/5):
                self.__mutation_rate /= 0.95
        else:
            self.__mutation_rate *= 2

    # copy data to self in order to return result
    self.data = gen[0].data
    return True # solution found
```

5.2 Methods

This class define multiple methods:

- **__clone() -> SudokuGenetic**
  copy a sudoku board in another instance

- **__rank_solution() -> int**
  get the fitness of the instance (the lowest the better, solution has rank 0)
- **__get_first_gen(size: int) -> List[SudokuGenetic]**
  get first generation of given size
- **__get_parent(candidates: List[SudokuGenetic]) -> SudokuGenetic**
  extract one parent from candidates
- **__generate(partner) -> Tuple[SudokuGenetic, SudokuGenetic])**
  get two child from self and parent
- **__mutate()**
  insert mutations in current instance