Programming Assignment #3 Report

0516007 高誌佑

1. Implementation Details

The KB is a two-dimensional list, and the KB0 is a one-dimensional list. (Each clause is a list)

Basically, the game flow is the same as mentioned in the spec, but there are some differences in my implementation:

1. There is no duplication and subsumption check in the matching phase. The duplication and subsumption check are done only in the insert phase. Since both check are done when there is a new clause to be inserted, the KB will always pass both check until there is a new clause. Then, just do both check in the insert phase.
2. For convenience, the stuck situation is detected when there is no single-literal clauses generated or marked in five continuous iterations.
3. Since I do not have enough time to do this assignment, there is no global constraints involved when generating clauses from the hints.

2. Experiments results

Below are the environments in my experiments:

- Machine (CPU): AMD Ryzen 7-3700X 3.6GHz
- Operating System: Ubuntu 18.04
- Language: **Python 3.7.7**

Success Rate and Execution Time Comparison

I do the experiments with three board sizes.

In each board size, I compare the results according to different numbers of initial safe cells and different numbers of mines.

For each setting, I run 100 games and count the number of success and the execution time. The execution time is the **average time of those successful games**.

The results are shown in the following tables.

- Easy level (Board Size: 9 x 9)

#Initial Safe Cells	#Mines	#Success	Execution Time
9	5	100	0.0248 s
9	10	83	0.0256 s
9	15	29	0.0274 s
15	5	100	0.0264 s
15	10	87	0.0260 s
15	15	45	0.0386 s
20	5	100	0.0280 s
20	10	90	0.0269 s
20	15	54	0.0325 s

- Medium level (Board Size: 16 x 16)

#Initial Safe Cells	#Mines	#Success	Execution Time
16	15	98	0.248 s
16	25	84	0.213 s
16	35	61	0.211 s
16	45	28	0.214 s
25	15	98	0.262 s
25	25	88	0.231 s
25	35	65	0.227 s
25	45	37	0.257 s
40	15	99	0.270 s
40	25	94	0.246 s
40	35	74	0.228 s
40	45	41	0.252 s

- Hard level (Board Size: 30 x 16)

#Initial Safe Cells	#Mines	#Success	Execution Time
22	45	90	0.716 s
22	60	66	0.652 s
22	75	33	0.658 s
22	99	0	-
50	45	92	0.748 s
50	60	77	0.701 s
50	75	37	0.681 s
50	99	4	1.007 s
100	45	92	0.859 s
100	60	84	0.765 s
100	75	64	0.787 s
100	99	10	1.313 s
150	45	95	0.933 s
150	60	87	0.878 s
150	75	70	0.890 s
150	99	42	1.361 s

3. Observations, and Interpretations

According to the experimental results, we can find that three levels have much in common.

1. Given the same number of initial safe cells, the success rate decreases as the number of mines grows. More mines means that there are less safe cells. Since the new clauses we get from the control module after marking a safe cell are the important clues in the game, it is hard to finish the game when the mines are too much.
2. Given the same number of mines, the success rate increases as the number of initial safe cells grows. More initial safe cells means that there are more clues in the beginning. It is more likely to finish the game with more clues.
3. In the case of the previous point, the execution time grows as the number of initial safe cells grows. Since the initial size of the KB is larger, the size will grow faster among the game. Larger KB means that matching, duplication check, and subsumption check takes more time, so it is reasonable that the execution time grows.

4. With the same number of mines, it is more likely to finish the game in the larger board. We can see the big difference of success rate in the following table. It is from the experimetal results of the medium level and the hard level. Even with more initial safe cells, the success rate is lower in medium level.
   I guess the reason is that the rate of the number of mines to the number of total cells is higher in the medium level. Since it is more difficult to mark a safe cell, it is harder to finish the game.

#Initial Safe Cells	#Mines	#Success	Board size
40	45	41	16 x 16 (Medium)
22	45	90	30 x 16 (Hard)

4. Extra discussions

1. Q: How to use first-order logic here?

   A: When generating clauses from hints, we can use first-order login instead.
   For example, the hint in one cell is 0, we can use the below notation.

$$\forall x \neg x, \text{ x is in } \{x_1, x_2, \dots\}$$

2. Q: Discuss whether forward chaining or backward chaining applicable to this problem.

   A: I think the forward chaining is applicable since we can use the known facts to get the goal step by step, and the backward chaining is not applicable since we do not know the goal in the beginning.

3. Q: Propose some ideas about how to improve the success rate of "guessing" when you want to proceed from a "stuck" game.

   A: Use the idea of the heuristic in Assignment#2.
   For example, use the Degree Heuristic to first guess the cell with the more constraints.

4. Q: Discuss ideas of modifying the method in Assignment#2 to solve the current problem.

   A: I have no idea.

5. Things I have learned

It is my first time to implement the logical inference techniques. Although the typical approach of the problem is more like the searching algorithm we use in the Assignment#2 and logical inference techniques are only applied to tiny toy problems, I still benefit a lot in this assignment. I never think about the concept of logical inference for AI this game. Thinking more with different concepts is always good to enhance myself.

6. Remaining Questions and Future Investigations

I do not implement the global constraints in this assignment. Next time, I will try to implement the global constraint as hints and find out the appropriate time to add the clauses from this hint. I will also think more about the extra discussions in the future.

Appendix: Source code

- main.py

```
import sys
import time
import random
import math
from itertools import combinations

direction = [(-1, -1), (-1, 0), (-1, 1), (0, -1), (0, 1), (1, -1), (1, 0), (1, 1)]

class Control():
    def __init__(self, board_size, num_mines):
        self.board_size = board_size
        self.num_mines = num_mines
        self.mines = [[False for _ in range(board_size[1])] for _ in range(board_size[0])]
        self.marked = [[-1 for _ in range(board_size[1])] for _ in range(board_size[0])]

    def initialize_board(self):
        coordinates = []
        for x in range(self.board_size[0]):
            for y in range(self.board_size[1]):
                coordinates.append((x, y))

        # sample the positions of mines
        mine_coordinates = random.sample(coordinates, self.num_mines)
        for coord in mine_coordinates:
            self.mines[coord[0]][coord[1]] = True

        print('Mines:')
        for row in self.mines:
            for col in row:
                if col:
                    print('*', end=' ')
                else:
                    print('-', end=' ')
            print('')

    def get_initial_safe_cells(self):
        # find all the safe cells
        coordinates = []
        for x in range(self.board_size[0]):
            for y in range(self.board_size[1]):
                if not self.mines[x][y]:
                    coordinates.append((x, y))

        num_safe_cells = round(math.sqrt(self.board_size[0] * self.board_size[1]))
        # num_safe_cells = 150
        return random.sample(coordinates, num_safe_cells)

    def get_unmarked_neighbors(self, x, y):
        # calculate the mines of unmarked neighbor cells
```

```
        num_mines = 0
        cells = []
        for d in direction:
            px = x + d[0]
            py = y + d[1]
            if self.is_outside(px, py) or self.marked[px][py] != -1:
                continue
            cells.append((px, py))
            if self.mines[px][py]:
                num_mines += 1

        return num_mines, cells

    def mark_cell(self, x, y, value):
        self.marked[x][y] = value

    def print_marked_cells(self):
        count = 0
        print('\nResults:')
        for row in self.marked:
            for col in row:
                if col == 1:
                    count += 1
                    print('*', end=' ')
                else:
                    print('-', end=' ')
            print('')
        print(f'Total mark {count} mines.')
        return count

    def is_outside(self, x, y):
        # check the given position is outside the board
        return (x < 0) or (x >= self.board_size[0]) or (y < 0) or (y >= self.board_size[1])


class Player():
    def __init__(self):
        self.kb = []
        self.kb0 = []

    def initialize_kb(self, safe_cells):
        # transfer the safe cells to single-literal clause and insert into KB
        for cell in safe_cells:
            cnf = [(cell[0], cell[1], False)]
            self.kb.append(cnf)

    def handle_single_literal_clause(self):
        cnf1 = self.kb.pop(0)
        self.kb0.append(cnf1[0])

        for cnf2 in self.kb:
            cnf = self.handle_resolution(list(cnf1), list(cnf2))
```

```
        self.insert_clause(cnf)

    return cnf1[0]

def handle_multiple_literal_clause(self):
    cnfs = []
    delete_index = []

    # pairwise matching
    for i, cnf1 in enumerate(self.kb):
        resolution = False
        for j, cnf2 in enumerate(self.kb[i + 1:]):
            if len(cnf1) > 2 or len(cnf2) > 2:
                continue
            cnf = self.handle_resolution(list(cnf1), list(cnf2))
            if len(cnf) > 0:
                resolution = True
                cnfs.append(cnf)

        if resolution:
            delete_index.append(i)

    for cnf in cnfs:
        self.insert_clause(cnf)
    for i in reversed(delete_index):
        del self.kb[i]

def find_complementary_pairs(self, cnf1, cnf2):
    # find out the pairs of complementary literals of two clauses
    pairs = []
    for i, l1 in enumerate(cnf1):
        for j, l2 in enumerate(cnf2):
            if l1[0] == l2[0] and l1[1] == l2[1] and l1[2] != l2[2]:
                pairs.append((i, j))
    return pairs

def handle_resolution(self, cnf1, cnf2):
    pairs = self.find_complementary_pairs(cnf1, cnf2)
    if len(pairs) != 1:
        return []

    # do the resolution to generate a new clause
    pair = pairs[0]
    del cnf1[pair[0]]
    del cnf2[pair[1]]
    cnf = cnf1 + cnf2
    return cnf

def generate_clause_from_hint(self, num_mines, cells):
    num_cells = len(cells)

    # generate clauses according to the situation
```

```
        if num_mines == 0:
            for cell in cells:
                cnf = [(cell[0], cell[1], False)]
                self.insert_clause(cnf)
        elif num_mines == num_cells:
            for cell in cells:
                cnf = [(cell[0], cell[1], True)]
                self.insert_clause(cnf)
        elif num_mines < num_cells:
            num = num_cells - num_mines + 1
            combs = list(combinations(cells, num))
            for comb in combs:
                cnf = []
                for cell in list(comb):
                    cnf.append((cell[0], cell[1], True))
                self.insert_clause(list(cnf))

            num = num_mines + 1
            combs = list(combinations(cells, num))
            for comb in combs:
                cnf = []
                for cell in list(comb):
                    cnf.append((cell[0], cell[1], False))
                self.insert_clause(list(cnf))

    def insert_clause(self, cnf):
        if len(cnf) == 0:
            return

        # do the resolution with all the clauses in KB0
        cnf.sort()
        updated_cnf = []
        for l1 in cnf:
            deleted = False
            for l2 in self.kb0:
                if l1[0] == l2[0] and l1[1] == l2[1] and l1[2] != l2[2]:
                    deleted = True
                    break
            if not deleted:
                updated_cnf.append(l1)

        if self.check_duplication(updated_cnf):
            return
        if self.check_subsumption(updated_cnf):
            return

        if len(updated_cnf) == 1:
            self.kb.insert(0, updated_cnf)
        else:
            self.kb.append(updated_cnf)

    def check_duplication(self, cnf1):
```

```
        if len(self.kb) == 0:
            return False

        for cnf2 in self.kb:
            if (len(cnf1) != len(cnf2)):
                return False
            for i in range(len(cnf1)):
                if cnf1[i] != cnf2[i]:
                    return False
        return True

    def check_subsumption(self, cnf1):
        kb = []
        res = False
        for cnf2 in self.kb:
            if len(cnf1) < len(cnf2) and self.check_strict(cnf1, cnf2):
                continue
            if len(cnf1) > len(cnf2) and self.check_strict(cnf2, cnf1):
                res = True
            kb.append(cnf2)

        if len(kb) < len(self.kb):
            self.kb = kb
        return res

    def check_strict(self, cnf1, cnf2):
        for l1 in cnf1:
            find = False
            for l2 in cnf2:
                if l1 == l2:
                    find = True
                    break

            if not find:
                return False

        return True

    def check_termination(self):
        return (len(self.kb) == 0)


class Game():
    def __init__(self, level):
        if level == 'Easy':
            self.board_size = (9, 9)
            self.num_mines = 10
        if level == 'Medium':
            self.board_size = (16, 16)
            self.num_mines = 25
        if level == 'Hard':
            self.board_size = (30, 16)
```

```python
            self.num_mines = 99
        self.control = Control(self.board_size, self.num_mines)
        self.player = Player()

    def start(self):
        self.control.initialize_board()
        safe_cells = self.control.get_initial_safe_cells()
        self.player.initialize_kb(safe_cells)

        stuck_count = 0
        while True:
            if self.player.check_termination() or stuck_count >= 5:
                return self.control.print_marked_cells() == self.num_mines

            # if there is a single-literal clause in KB
            if len(self.player.kb[0]) == 1:
                stuck_count = 0
                # print('case1')
                cell = self.player.handle_single_literal_clause()
                self.control.mark_cell(cell[0], cell[1], int(cell[2]))
                if cell[2] == False:
                    num_mines, cells = self.control.get_unmarked_neighbors(cell[0], cell[1])
                    self.player.generate_clause_from_hint(num_mines, cells)
            else:
                # print('case2')
                stuck_count += 1
                self.player.handle_multiple_literal_clause()


if __name__ == '__main__':
    if len(sys.argv) != 2:
        print('Usage: python main.py [Easy|Medium|hard]')
        sys.exit()

    success_count = 0
    total_time = 0.0
    for i in range(100):
        start = time.time()
        game = Game(sys.argv[1])
        success = game.start()
        end = time.time()
        if success:
            total_time += end - start
            success_count += 1

    print(f'Success count: {success_count}')
    if success_count > 0:
        print(f'Average time: {total_time / success_count} seconds')
```