CS5011 Assignment 2: Logic

Student ID: 210034146
(Dated: March 14, 2022)

1: INTRODUCTION

1.1: Sections implemented

The following list describes the sections implemented in this assignment:

- Part 1 (Basic Obscured Sweeper Agent): Attempted and fully working.
- Part 2 (Beginner Obscured Sweeper Agent): Attempted and fully working.
- Part 3 (Intermediate Obscured Sweeper Agent): Attempted and fully working logic agent which uses the logicNG library to test DNF-KBU for satisfiability.
- Part 4 (Intermediate Obscured Sweeper Agent 2): Attempted and fully working logic agent which invovles directly encoding the KBU in CNF form.

1.2: Running instructions

Steps to run the algorithm:

- Navigate to the src folder using CDM/terminal/command line.
- Type the following:
  ./playSweeper.sh <Agent> $< WorldID >$

1.4: Instructions: Stacscheck

The stacscheck is expanded to validate agents p3 and p4. Furthermore, additional worlds are also added to validate the correctness of the system. The modified stacscheck can be run by following the steps below.

- Using CDM/terminal/command line navigate to the directory which contains the src and the Tests folders.
- Type the following to run the tests:
  stacscheck $<$ test directory path $>$
- For example, in this case the tests folder is located in the starterCode directory (same directory as the src folder). Type the following:
  stacscheck /Tests

2: DESIGN AND IMPLEMENTATION

2.1: Brief introduction to logic based agents

Logic-based agents in Artificial Intelligence represent the structure of their internal state (partially observable world) in a container of information known as a knowledge base. The agent can update this knowledge base by observing or interacting with the world. Therefore, the agent can use logic to make inferences about what action the agent can take to uncover the hidden states (states outside of the partially observable world). The logic agent can infer hidden states by turning the query (question about the hidden state/model) into a satisfiability problem. If the knowledge base entails this query (α), then the query is added to the knowledge base.

$$KB| = A \text{ iff } KB \wedge \neg A \text{ is unsatisfiable} \tag{1}$$

2.2: Main Design

The problem at hand involves solving a game of obscured minesweeper represented by a 2D board of NxN cells (N is the length of the map in one dimension). Each cell could be either one of three things; a mine, a blocked cell, or a clue cell (see figure 1). The agent loses the game if a mine cell is probed. A blocked cell is inaccessible, and when a clue cell is probed, the clue number $clue$ represents the number of mines in the neighboring adjacent cells ($0 \leq clue \leq 8$). The agent's view of the world (partial view) is distinct from the real world, which contains the absolute information of all the cells. The agent is aware of the total number of mines in a board and the locations of all blocked cells. Furthermore, the the agent is aware that the top-left cell (**i.e.** coordinate [0,0]), and the cell in the center of the board (**i.e.** coordinate $[\frac{boardlength-1}{2}, \frac{boardlength-1}{2}]$) are always safe to probe.

Initially, the agent's world map is represented by unmarked cells (denoted by '?') and blocked cells (denoted by 'b'). Once all safe cells have been probed successfully, and all mine cells have been marked with an asterisk ('*'), the agent wins the game. The agent loses the game if any mine cell is probed. Furthermore, the game could cascade into a state where no new information about the unmarked states could be harnessed; in such a scenario, the agent neither wins nor loses the game. The algorithm starts by defining cells into states; this is done by labeling each cell in the map with a number n. For a NxN board, there are a total of N^2 states. Each cell is assigned a number between $0 \leq n < N^2$ based on its position. For example in a 3x3 map, cells (0,0), (1,1), and (2,2) are mapped into the states 0, 4, and 8 respectively.

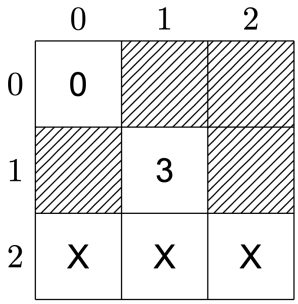

FIG. 1: Example Obscured Sweeper World, TEST1. The shaded areas are blocked cells while the cells marked with 'X' are cells containing mines. Figure taken from [1]

As mentioned before, the real board and the agent's board are maintained separately. If a cell is probed, the cell needs to be uncovered. The uncovering process involves extracting the nature (value) of the cell from the truth board. This is done using the *mapper* function described in listing one.

```
function mapper(state, truth_board):
  state_cooordinate = state_to_coordinate(
    truth_board.length,state)
  truth_value = board[state_cooordinate[0]][
    state_cooordinate[1]]
  return cell
```

Listing 1: Mapper function which returns the value of a cell once it has been probed.

Furthermore, three out of the four agents make inferences by taking the neighboring cells into account. The *neighbours* function described in listing 2 returns a list of neighboring states to the current state.

```
function neighbours(current_state, current_board):
  // converting state to coordinates
  temp_cord = state_to_coordinate(current_board.
    length, current_state);
  neighbours = new list
  // finding neighbours using a nested for loop
  for (int i = temp_cord[0]-1; i < temp_cord[0]+2;i
    ++):
    for (int j = temp_cord[1]-1; j <temp_cord[1]+2;
    j++):
    // ensuring the neighbours exist within the
    board
      if (!(i < 0 || i > (current_board.length -1)
    || j < 0 || j > (current_board.length -1) )):
          neighbour = coordinate_to_state(
    current_board.length,new int[]{i,j}):
          if (neighbour != temp_state):
            neighbours.add(neighbour)
  return neighbours
```

Listing 2: Neighbours function returns a list of neighbouring states to the current state.

A dictionary (java hashmap) is used to maintain the agent's view of the world. The map contains keys which are the state labels ($0 \leq n < N^2$), and the values are the deciphered values of the states (**i.e.** the clue number, danger state, blocked state).

The map is initially filled with keys that represent the blocked state. This is represented in listing 3.

```
HashMap map = new HashMap()
for(int i = 0;i<truth_board.length**2;i++):
  state_label = mapper(i, truth_board);
  // adding blocked cells to the map.
  if (state_label == 'b'):
    map.put(i,'b');
```

Listing 3: Adding blocked states to the agents knowledge base/map.

As mentioned before, this submission contains four agents that use various techniques to solve the game. The following section of the report gives an overview of these different agents and their respective strategies.

3: AGENTS

3.1: Basic Obscured Sweeper Agent

The basic sweeper agent does not incorporate any logic or strategy. Given an unmarked board, the agent probes each cell in the board in order using a for loop which iterates over states in ascending order (**i.e.** left to right on each row; top row to bottom row). Each probed state is added to the agent's hashmap (knowledge base) using the mapper function described in listing 1, and the agent's board is updated to reflect this. The agent loses the game if it probes a mine cell (in which case the probed mine cell on the agent's board is denoted by '-'). The agent wins the game if the number of unprobed cells equals the number of mines (**i.e.** all safe cells have been probed). Listing 4 describes the pseudo-code implementation for this agent.

```
// each cell is given a number between 0 to
    board_size-1
for(i = 0; i < board_size;i++):
    state_value = mapper(i, truth_board);

    if (!map.containsKey(i)):
      // adding non probed cells to the map.
      map.put(i,temp_label);

      if (temp_label == '0'):
          // neighbouring cells are uncovered if
    the current cell = '0'
        map_neighbours(i,board,map);

      if (temp_label == 'm'):
        game_lost = true;
        break;

      if (map.size() == (board_size - mine_size)):
        game_won = true;
        break;
```

Listing 4: Basic Obscured Sweeper implementation.

3.2: Beginner Obscured Sweeper Agent

The beginner obscured sweeper agent plays the game by implementing the single-point reasoning strategy (SPS). This strategy works by first probing the safe cells in order to uncover clues and also potentially uncovering the neighbors of the safe cells if either of the safe cells equals '0', which could potentially trigger a cascading uncovering process.

The SPS strategy works by iterating through each unmarked cell ('?') on the board. Given an unmarked cell, each of its neighboring safe cells (cells with clues) are examined to see if they satisfy one of two conditions:

- Condition 1 (All marked neighbours): For a given safe neighbour k, if the clue (value of the cell $-$ # danger neighbours) equals the number of neighbours of k which are unmarked, then all unmarked neighbours of k are marked as danger (denoted by '*') and added to the knowledge base (see figure 2).

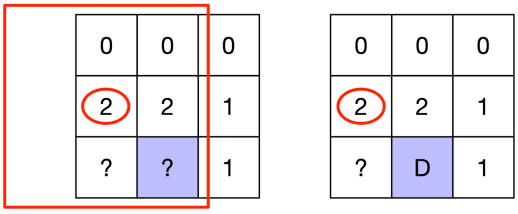

FIG. 2: All marked neighbours. Figure taken from [2]

- Condition 2 (All free neighbours): For a given safe neighbour k, if the value of the cell equals the number of neighbours of k which are already marked as danger ('*'), then all unmarked neighbours of k are uncovered (using the mapper function) and added to the knowledge base (see figure 3).

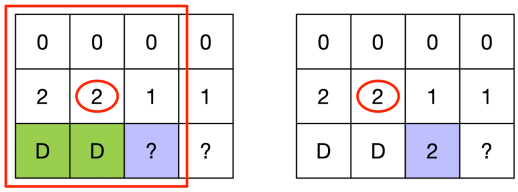

FIG. 3: All free neighbours. Figure taken from [2]

Furthermore, this process is repeated over all unmarked cells until the agent wins the game or the agent reaches a point where no new information is being added via the SPS strategy (# unmarked cells do not change after a point). If all safe cells have been probed and all mines have been marked, the agent wins the game. The agent cannot lose the game since, if implemented correctly, the SPS strategy would never probe a mine. The game is left in a 'not terminated' state if the agent cannot probe/mark any remaining unmarked cells. The beginner obscured sweeper agent has a separate class of its own (P2.java), and it is invoked in the A2main.java script by calling the P2.solver function.

3.3: Intermediate Obscured Sweeper Agent

The intermediate obscured sweeper agent is made up of two parts. The first part uses propositional logic written in DNF (disjunctive normal form/disjunction of conjunctions) in order to identify which unmarked cells are safe and hence can be probed. The second part of this agent is the same beginner obscured sweeper agent that uses the SPS strategy to mark and uncover cells.

Like the beginner obscured sweeper agent, the intermediate obscured sweeper agent starts by first probing the safe cells in order to find clues and uncover other neighbouring cells. The logic part of this agent involves probing every safe cell s whose neighbouring cell/cells are unmarked. Using the clue (value of the cell $-$ # danger neighbours) of cell s, the logic agent then creates a sentence using the sentenceConstructorDNF function which describes all possible combinations (mine vs not mine) of unmarked neighbours of s. This sentence is added to the KBU (knowledge base of unknowns), and the process is repeated for every safe cell s. Where all sentences undergo disjunction ($\vee$) with each other. For example, if state 4 (with clue == 1) had two unmarked neighbours; 2,5. Then the sentenceConstructorDNF function would create the following sentence.

$$\text{sentence}_{state4} = (2\& \sim 5)|(\sim 2\&5) \tag{2}$$

Where the numbers represent the state, '$\sim$' represents negation (**i.e.** ~ 2 implies the state 2 is not a mine), '|' represents disjunction and '&' represents conjunction.

Once all safe cells s have been probed, the KBU along with the query (whether an unmarked state is a mine) is parsed into the logicNG library, which tests if the query is satisfiable (see equation 3) using the proveMineorFreeDNF function. If the query is unsatisfiable (output == false), then the unmarked state in question is probed and added to the map. This satisfiability test is repeated for every unmarked cell on the board.

$$KB| = \neg\text{state iff } (KB \wedge \text{state}) \text{ is unsatisfiable} \tag{3}$$

This agent switches between the DNFlogic and SPS implementation until the agent wins the game or the agent reaches a point where no new information is being added via both strategies (# unmarked cells do not change after a point). The agent wins the game if all safe cells have been probed and all mines have been marked. The agent cannot lose the game since, if implemented correctly, the logic+SPS strategy would never probe a mine. The game is left in a 'not terminated' state if the agent is unable to probe/mark any remaining unmarked cells. The DNFlogic agent has a separate class of its own (P3.java), and it is invoked in the A2main.java script by calling the P3.solver function.

3.3: Intermediate Obscured Sweeper Agent 2

The intermediate obscured sweeper agent 2 is also is made up of two parts. The first part uses propositional logic written in CNF (Conjunctive normal form/conjunction of disjunctions) in order to identify which unmarked cells are safe and hence can be probed. The second part of this agent is the same beginner obscured sweeper agent that uses the SPS strategy to mark and uncover cells. The intermediate obscured sweeper agent 2 involves writing logic sentences in CNF form, which is then transformed into DIMACS format before feeding it into the SAT solver.

The logic part of this agent involves probing every safe cell s whose neighbouring cell/cells are unmarked. Using the clue (value of the cell $-$ # danger neighbours) of cell s, the logic agent then creates a sentence using the sentenceConstructorCNF function which describes CNF combinations (mine vs not mine) of unmarked neighbours of s. This sentence is added to the KBU, and the process is repeated for every safe cell s. Where all sentences are converted into clauses (defined by separate arrays) which represent conjunction ($\wedge$). For example, if state 4 (with clue == 1) had two unmarked neighbours; 2,5. Then the sentenceConstructorCNF function would create the following two clauses.

$$\text{sentence}_{state4} = [[2, 5], [-2, -5]] \tag{4}$$

Where each 1d list within the 2d array represents a clause (in conjunction ($\wedge$) with the next clause). The numbers represent the states, and the numbers separated by a comma ',' within a clause indicate disjunction ($\vee$) between literals, the negative numbers '$-$' represent negation (**i.e.** -2 implies the state 2 is not a mine).

Once all safe cells s have been probed, the KBU along with the query (whether an unmarked state is a mine) is parsed into the sat4j's ISolver, which tests if the query is satisfiable (see equation 3) using the proveMineorFreeCNF function. If the query is unsatisfiable (output == false), then the unmarked state in question is probed and added to the map. This satisfiability test is repeated for every unmarked cell on the board.

The algorithm switches between the CNFlogic and SPS implementation until the agent wins the game or the agent reaches a point where no new information is being added via both strategies (# unmarked cells do not change after a point). The agent wins the game if all safe cells have been probed and all mines have been marked. The agent cannot lose the game since, if implemented correctly, the logic+SPS strategy would never probe a mine. The game is left in a 'not terminated' state if the agent is unable to probe/mark any remaining unmarked cells. The CNFlogic agent has a separate class of its own (P4.java), and it is invoked in the A2main.java script by calling the P4.solver function.

3: TEST SUMMARY

The system passes 19 out of 19 stacscheck test cases provided on studres. However, the stacscheck only validates the P1 and P2 agents. For this reason, the stacscheck was expanded to also validate P3 and P4 agents (where the expected outputs were calculated by hand). Furthermore, all four agents were tested on additional maps, which resulted in a total of 51 test cases. The system passes 51 out of 51 stacscheck test cases.

3.1: Working example

A working example of all agents on test case 3 (3x3 board) is demonstrated in the following subsections. Listing 5 depicts the 2d map belonging to test case 3. The first board is the real board, the second board is the agent's initial view of the world, and the last board represents the state numbers (each cell is assigned a number n where $(0 \leq n < 3^2)$).

```
//Truth board
        0 1 2
        - - -
    0| 0 0 b
    1| 1 1 1
    2| 1 m 1

//Agent's board
        0 1 2
        - - -
    0| ? ? b
    1| ? ? ?
    2| ? ? ?

//State numbers
        0 1 2
        - - -
    0| 0 1 2
    1| 3 4 5
    2| 6 7 8
```

Listing 5: Obscured sweeper board.

3.1.2: Agent: P1

The P1 agent works by sweeping through every cell (left to right on each row; top row to bottom row). The following list describes each step taken by the P1 agent.

- Agent probes state 0, which uncovers the neighbouring states 1,3,4, and 5.
- The agent now directly probes state 6 since other states have been uncovered.
- The agent probes state 7, which is a mine. State 7 is marked with '-' on the agent's board. The agent loses the game.

FIG. 4: System output: P1 agent

FIG. 5: System output: P2 agent

3.1.2: Agent: P2

The P2 agent works by using the SPS strategy. The following list describes each step taken by the P2 agent.

- Agent probes state 0, which uncovers the neighbouring states 1,3,4, and 5.
- The agent now examines the neighbours of the unmarked state 6. None of the neighbours of state 6 satisfy AFN and AMN conditions.
- The agent now examines the neighbours of the unmarked state 7. None of the neighbours of state 7 satisfy AFN and AMN conditions.
- The agent now examines the neighbours of the unmarked state 8. None of the neighbours of state 8 satisfy AFN and AMN conditions.
- The agent comes to a halt since the strategy ceases to provide any more logical information. The game ends in a non-terminated state.

3.1.3: Agent: P3

The P3 agent works by using a combination of logic and SPS. The following list describes each step taken by the P3 agent.

- Logic agent probes state 0, which uncovers the neighbouring states 1,3,4, and 5.
- The agent now examines state 3 (clue = 1) whose neighbours (states 6 and 7) are unmarked. The following DNF sentence is added to the KBU:
  $(6\& \sim 7)|(7\& \sim 6)$.
- The agent now examines state 4 (clue = 1) whose neighbours (states 6, 7, and 8) are unmarked. The following DNF sentence is added to the KBU:
  $(6\& \sim 7\& \sim 8)|(7\& \sim 6\& \sim 8)|(8\& \sim 6\& \sim 7)$.
- The agent now examines state 5 (clue = 1) whose neighbours (states 7, 8) are unmarked. The following DNF sentence is added to the KBU:
  $(7\& \sim 8)|(8\& \sim 7)$.
- The agent uses the logicNG library to infer if the unknown states are safe to probe. States 6 and 8 are proven to be safe and hence are probed and added to the agent's board.
- The agent falls back to the SPS strategy, which probes the only remaining unmasked state, state 7. State 3 (neighbour of state 7) satisfies the AMN condition and state 7 is marked as danger '*'.
- The agent wins the game.

3.1.4: Agent: P4

The P4 agent works by using a combination of logic and SPS. The following list describes each step taken by the P4 agent.

- Logic agent probes state 0, which uncovers the neighbouring states 1,3,4, and 5.
- The agent now examines state 3 (clue = 1) whose neighbours (states 6 and 7) are unmarked. The following

```
----------------------------------------

Agent P3 plays TEST3


    0 1 2
    - - -
 0| 0 0 b
 1| 1 1 1
 2| 1 m 1

Start!
Final map

    0 1 2
    - - -
 0| 0 0 b
 1| 1 1 1
 2| 1 * 1


Result: Agent alive: all solved
```

FIG. 6: System output: P3 agent

```
----------------------------------------

Agent P4 plays TEST3


    0 1 2
    - - -
 0| 0 0 b
 1| 1 1 1
 2| 1 m 1

Start!
Final map

    0 1 2
    - - -
 0| 0 0 b
 1| 1 1 1
 2| 1 * 1


Result: Agent alive: all solved
```

FIG. 7: System output: P4 agent

CNF clauses are added to the KBU:
$[[6, 7], [-6, -7]]$.

- The agent now examines state 4 (clue = 1) whose neighbours (states 6, 7, and 8) are unmarked. The following CNF clauses are added to the KBU:
$[[-6, -7], [-6, -8], [-7, -8], [6, 7, 8]]$.

- The agent now examines state 5 (clue = 1) whose neighbours (states 7, 8) are unmarked. The following CNF clauses are added to the KBU:
$[[7, 8], [-7, -8]]$.

- The agent uses the sat4j library to infer if the unknown states are safe to probe. States 6 and 8 are proven to be safe and hence are probed and added to the map.

- The agent falls back to the SPS strategy, which probes the only remaining unmasked state, state 7. State 3 (neighbour of state 7) satisfies the AMN condition and state 7 is marked as danger '*'.

- The agent wins the game.

4: EVALUATION AND CONCLUSION

The fraction of covered cells γ at the end of the game is used as a metric to compare the performance of each agent. The value of γ for each agent increases with increasing board size. Across all board sizes (3x3,5x5,7x7,9x9), the average value of γ for each agent is depicted in figure 8. Figures 9, 10, 11, 12 represent γ for each board size. Considering the average γ across all board sizes and also γ for each board size, the logic-based agents (P3 and P4) clearly outperform the SPS agent (P2), which in turn surpasses the basic sweeper agent (P1). The sweeper agent performs relatively poorly since it is highly likely that a sweeper agent loses the game (by running into a mine) before it has had a chance to explore a fair share of the map. The sweeper agent leaves more than half of the cells covered (unexplored) across most board sizes. The logic bases agents, on average, leave half of the cells unprobed compared to the SPS agent. Assuming the correctness of the system holds, logic bases agents should be used to play the minesweeper game since they far outperform any other agent type.

REFERENCES

[1] *CS5011: A2 - Logic - Obscured Sweeper*, School of Computer Science, University of St Andrews, (2022)

[2] *CS5011 Logical Agents - The Danger Sweeper* , M.S Chang, School of Computer Science, University of St Andrews, (2022)

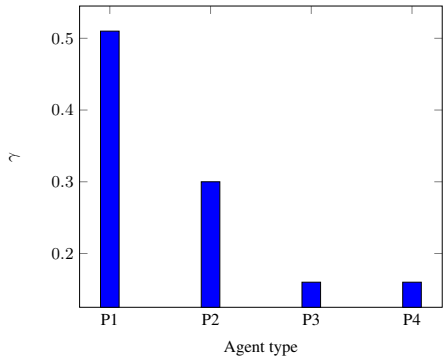

FIG. 8: Average fraction of covered cells at the end of the game for all board sizes.

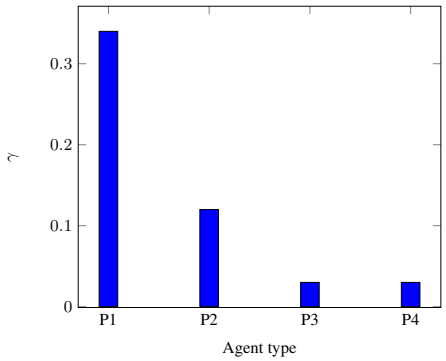

FIG. 9: Fraction of covered cells at the end of the game for a 3 x 3 board.

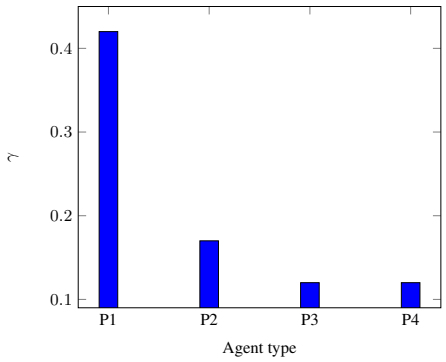

FIG. 10: Fraction of covered cells at the end of the game for a 5 x 5 board.

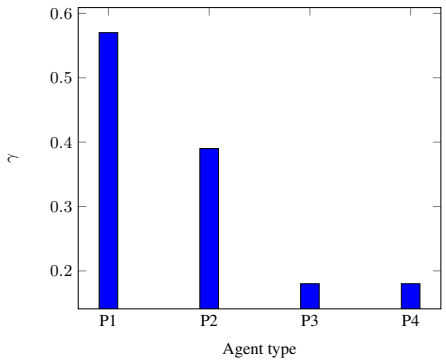

FIG. 11: Fraction of covered cells at the end of the game for a 7 x 7 board.

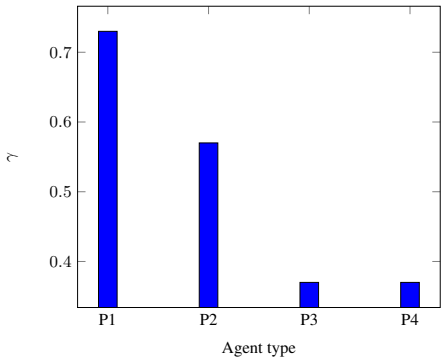

FIG. 12: Fraction of covered cells at the end of the game for a 9 x 9 board.