University of St Andrews

Assignment 3

Logical Agents

Student ID: 120022067

University of St Andrews

CS5011 Artificial Intelligence Practice

Contents

List of Figures

List of Tables

Listings

1 Introduction

The nettle sweeper game is inspired by the Mineswepeer computer game. The world in nettle sweeper game is a square grid of NxN squares with M nettles scattered among them. The squares can be uncovered in any order. If the uncovered square contains a nettle, the game is over. Otherwise, a number appears that indicates the number of nettles in the 8 adjacent squares.

The aim of this assignment is to implement an AI logical agent that is able to play and solve a nettle sweeper game.

1.1 Parts Implemented

Part 1: Implemented every feature stated in the requirements. The agent in this part can use Single Point Strategy to uncover/mark cell. The agent resorts to random guessing when no other move can be made. Full details of the implementation can be found in Section 3.

Part 2: For part 2, the agent can use Easy Equation Strategy and Single Point Strategy to uncover/mark cell. The agent resorts to random guessing when no other move can be made. Full details of the implementation can be found in Section 4.

Part 3: The agent in this part is able to use DPLL solver to prove that a given cell does or does not contain a nettle. An in-depth discussion of part 3's implementation can be found in Section 5.

1.2 Library

This section covers the third library used in this assignment.

1.2.1 AIMA Core

The implementation of the DPLL algorithm in this library is used provide satisfiability information for the agent in part 3 (aimacode 2017).

2 Literature Review

2.1 Minesweeper

Minesweeper is a game where mines are hidden within a grid of squares (Donner & Johnson 2013). The player starts the game with an n x m rectangular grid of covered squares (Becerra 2015). At each turn, the player can uncover a square revealing either a mine or a number (Becerra 2015). Becerra (2015) says that this number indicates the number of mines adjacent to that particular square. The number ranges from 0 to 8 because a square cannot have more than eight neighbors. An illustration of Minesweeper game can be found

in Figure 1. The game is over when the player uncover a cell containing a mine (Becerra 2015). The goal is to uncover all safe squares in the quickest time possible (Donner & Johnson 2013). According to Donner & Johnson (2013), the game became famous when Microsoft included it with Windows 3.1.

Figure 1: Minesweeper game.

2.2 Version History

According to Donner & Johnson (2013), there is very little information about the early Beta versions of Minesweeper. Donner & Johnson (2013) states that the game's original title was 'Mine'.

2.2.1 Mine 2.9

This version was created in 1990 (Donner & Johnson 2013). It has the same standard rules as Minesweeper. The game has three difficulty levels: Beginner (8x8, 10 mines), Intermediate (16x16, 40 mines) and Expert (24x24, 99 mines) (Donner & Johnson 2013). Donner & Johnson (2013) says that the game used bomb graphics instead of mines. Some screenshots of Mine2.9 can be found in Figure 2.

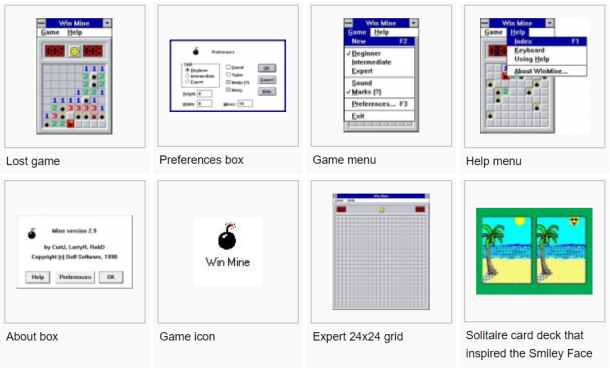


Figure 2: Screenshots of Mine2.9 (Donner & Johnson 2013).

2.2.2 Windows Entertainment Pack

Donner & Johnson (2013) states that Microsoft first released Minesweeper as part of the Windows Entertainment Pack (WEP) on 8 Oct 1990. The graphics for mines and flags were updated and Expert level changed to a 1630 grid (Donner & Johnson 2013). The game icon also changed to mine icon (Donner & Johnson 2013). There are some changes to the help file, and the game was officially named Minesweeper for the first time (Donner & Johnson 2013). Some screenshots of WEP version of Minesweeper is shown in Figure 3.

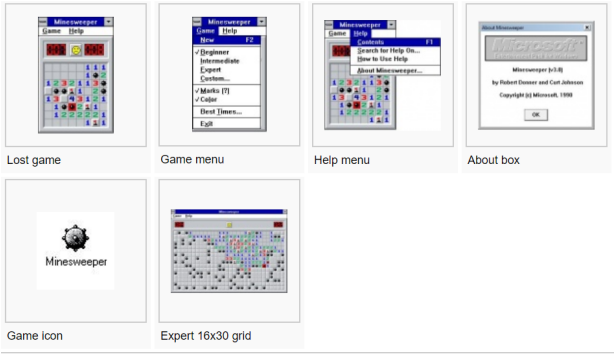


Figure 3: Screenshots of WEP version of Minesweeper (Donner & Johnson 2013).

2.2.3 Windows 3.1

This version was released on 6 April 1992 (Donner & Johnson 2013). The changes in this version include new Help file and storing Highscores in editable Winmine.ini file in the Windows folder (Donner & Johnson 2013). Some screenshots of Windows 3.1 version of Minesweeper is shown in Figure 4

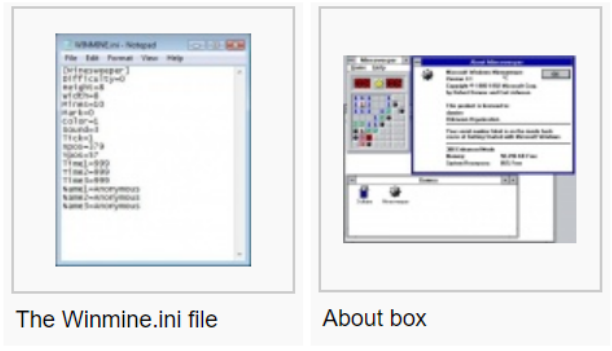


Figure 4: Screenshots of Windows 3.1 version (Donner & Johnson 2013).

2.2.4 Windows 95

This version was released on 24 Aug 1995 (Donner & Johnson 2013). In this version, the graphics of the game were cleaned up (Donner & Johnson 2013). Donner & Johnson (2013) says that the Help file was also more condensed. Some screenshots of Windows 95 version of Minesweeper is shown in Figure 5.

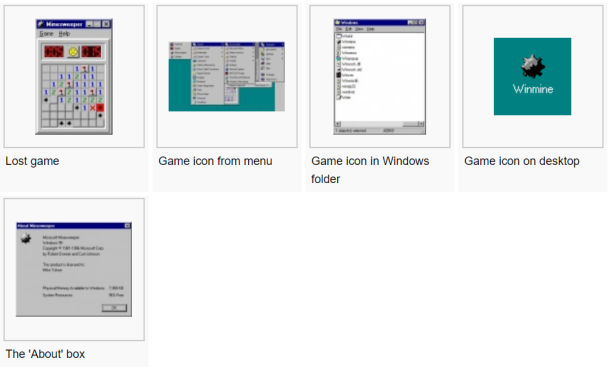


Figure 5: Screenshots of Windows 95 version (Donner & Johnson 2013).

2.2.5 Windows 98

According to Donner & Johnson (2013), the only change to this version was a new Help file.

2.2.6 Windows 2000

This version was released on 17 February 2000 (Donner & Johnson 2013). Donner & Johnson (2013) states that four major changes to the game are:

1. The Beginner grid size was increased from 8x8 to 9x9, but the number of mines did not change.
2. The Timer Jump bug was removed.
3. Moving Window bug was removed.
4. Winmine.ini file was removed. The game values were stored directly in the Registry.

Some screenshots of Windows 2000 version of Minesweeper is shown in Figure 6.

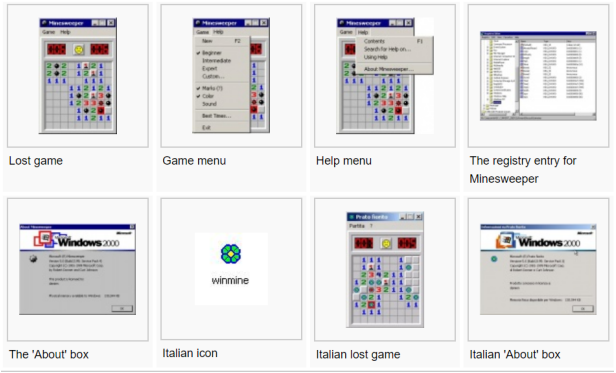


Figure 6: Screenshots of Windows 2000 version (Donner & Johnson 2013).

2.2.7 Windows ME

This version was released on 14 September 2000 (Donner & Johnson 2013). This version has the same beginner grid as Windows 98 version, but the Help file is slightly different (Donner & Johnson 2013). Some screenshots of Windows ME version of Minesweeper is shown in Figure 7.

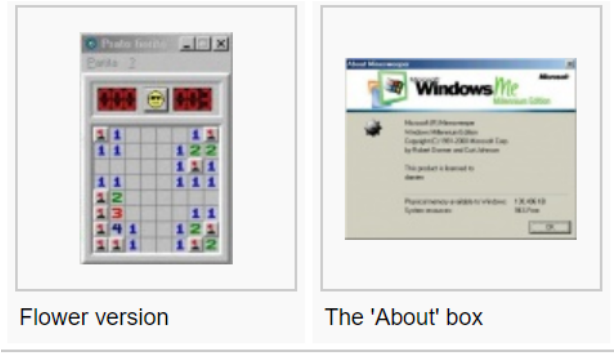


Figure 7: Screenshots of Windows ME version (Donner & Johnson 2013).

2.2.8 Windows XP

This version was released on 25 October 2001 (Donner & Johnson 2013). The changes to this version (from 2000) were a new icon and a new Help file (Donner & Johnson 2013). Some screenshots of Windows XP version of Minesweeper is shown in Figure 8.

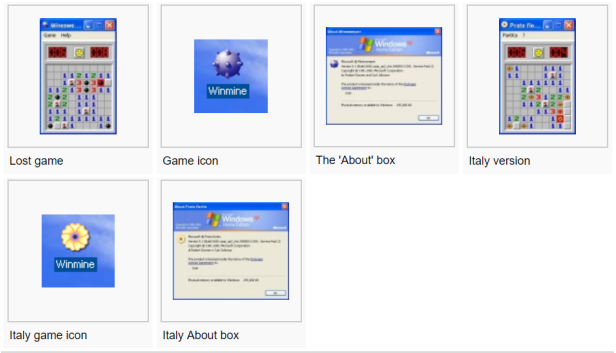


Figure 8: Screenshots of Windows XP version (Donner & Johnson 2013).

2.2.9 Windows Vista

This version was released on 30 January 2007 (Donner & Johnson 2013). According to Donner & Johnson (2013), Minesweeper was completely redesigned by Oberon Media to have new graphics and sound effects. Some screenshots of Windows Vista version of Minesweeper is shown in Figure 9.

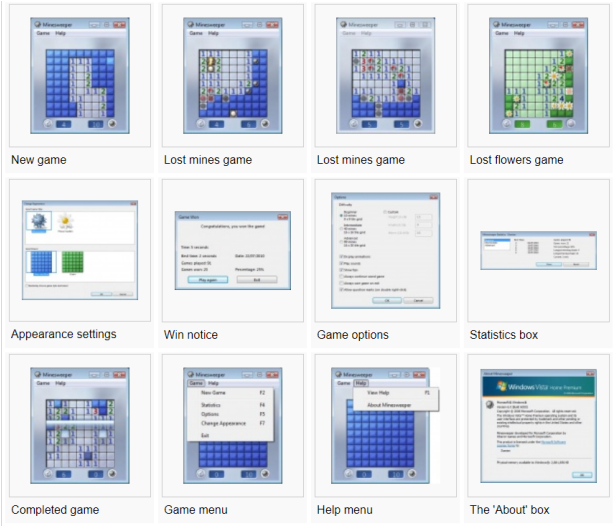


Figure 9: Screenshots of Windows Vista version (Donner & Johnson 2013).

2.2.10 Windows 7

This version was released on 22 October 2009 (Donner & Johnson 2013). The changes to this version were Help menu and Help file (Donner & Johnson 2013). Some screenshots can be found in Figure 10.

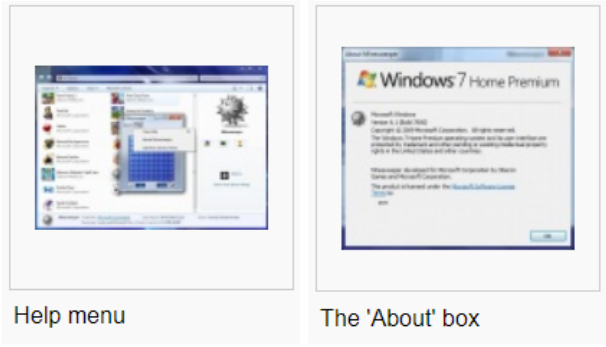


Figure 10: Screenshots of Windows 7 version (Donner & Johnson 2013).

3 Part 1

3.1 Design

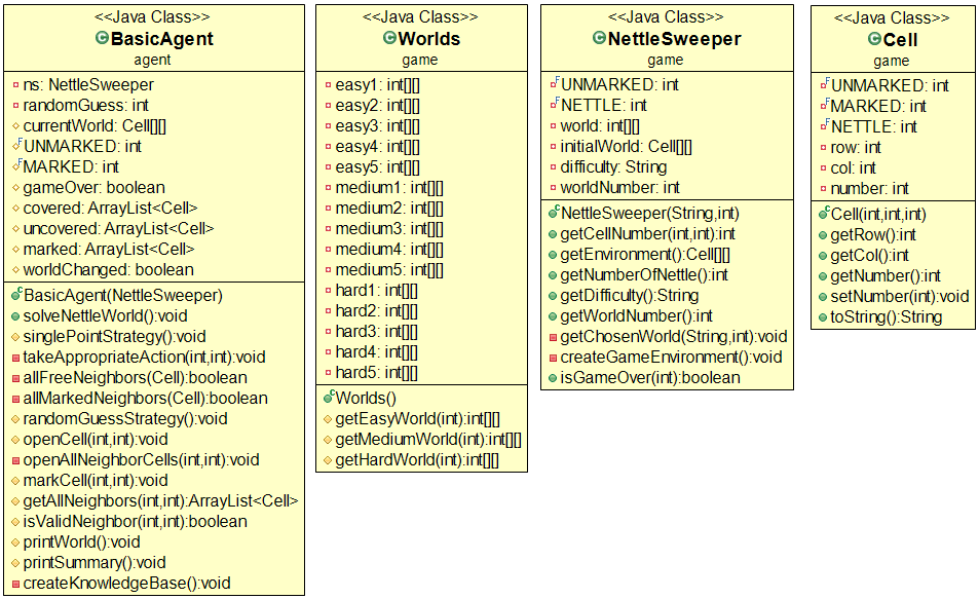


Figure 11: Class diagram for part 1.

3.1.1 PEAS Agent Model

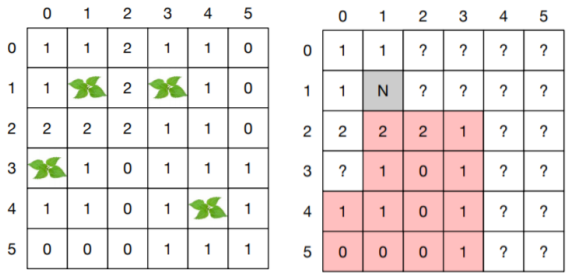


Figure 12: **Left**: the game's view of the nettle world. **Right:** the agent's view of the nettle world (Toniolo 2017*c*)

Performance measure: Random guessing required to solve each of the nettle worlds. This information is stored in ***randomGuess*** attribute of ***BasicAgent*** class as shown in Figure 11.

Environment: A square grid of NxN squares with M nettles scattered among them. The agent retrieves the initial environment with ***getEnvironment*** method of ***NettleSweeper*** class (see Figure 11). The environment is then saved in ***currentWorld*** attribute of ***BasicAgent*** class. The view of the environment is illustrated in Figure 12.

Actuator: The actions that the agent can perform are uncover cell and mark cell as nettle. The methods that are used as actuators are ***openCell()*** and ***markCell()***. These methods can be found in ***BasicAgent*** class (see Figure 11).

Sensor: The agent gets the percepts from the environment. These include the cells that are yet to be uncovered, the cells already uncovered, the cells that are marked as nettles, and total number of nettles. The percepts are stored in ***covered, uncovered, marked, and totalNettle*** attributes (knowledge base) of ***BasicAgent*** class (see Figure 11).

3.1.2 Game Infrastructure

```
public boolean isGameOver(int cellNumber) {
    if (cellNumber == NETTLE) {
      return true;
    }
    return false;
  }
```

Listing 1: The code used to inform the agent if the game is over or not.

The implementation of the game infrastructure is located in the ***game*** package. The core of the game can be found in ***NettleSweeper*** class. It contains a nettle sweeper world selected by the user, total number of nettles and the number behind each cell. It can inform the agent when the game is lost using ***isGameOver()*** method (see Listing 1). It is also responsible for creating the initial game environment for the agent with the method ***createGameEnvironment()*** (see Listing 2).

```
private void createGameEnvironment() {
    // create game world for the agent
    // all cells are covered at the beginning
    initialWorld = new Cell[world.length][world.length];
    for (int i = 0; i < world.length; i++) {
      for (int j = 0;j < world.length;j++) {
        initialWorld[i][j] = new Cell(i, j, UNMARKED);
      }
    }
  }
```

Listing 2: The code used to create initial state of the game for the agent.

The ***Worlds*** class contains all the nettle worlds provided. They are in 2D array of integers format (see Figure 11). The world chosen by the user will be loaded into ***world*** attribute of ***NettleWorld*** class. The code that is used to load the chosen world is shown in Listing 3.

```
private void getChosenWorld(String difficulty, int worldNumber) {
    // get the nettle world based on difficulty and world number
    Worlds w = new Worlds();
    if (difficulty.equals("easy")) {
      world = w.getEasyWorld(worldNumber);
    } else if (difficulty.equals("medium")) {
      world = w.getMediumWorld(worldNumber);
    } else {
      world = w.getHardWorld(worldNumber);
    }
  }
```

Listing 3: The code used to load the chosen map into the game.

The ***Cell*** class represents each cell that the agent interacts with in the nettle world. It contains the row number, column number, and number of a cell.

3.1.3 Agent Implementation

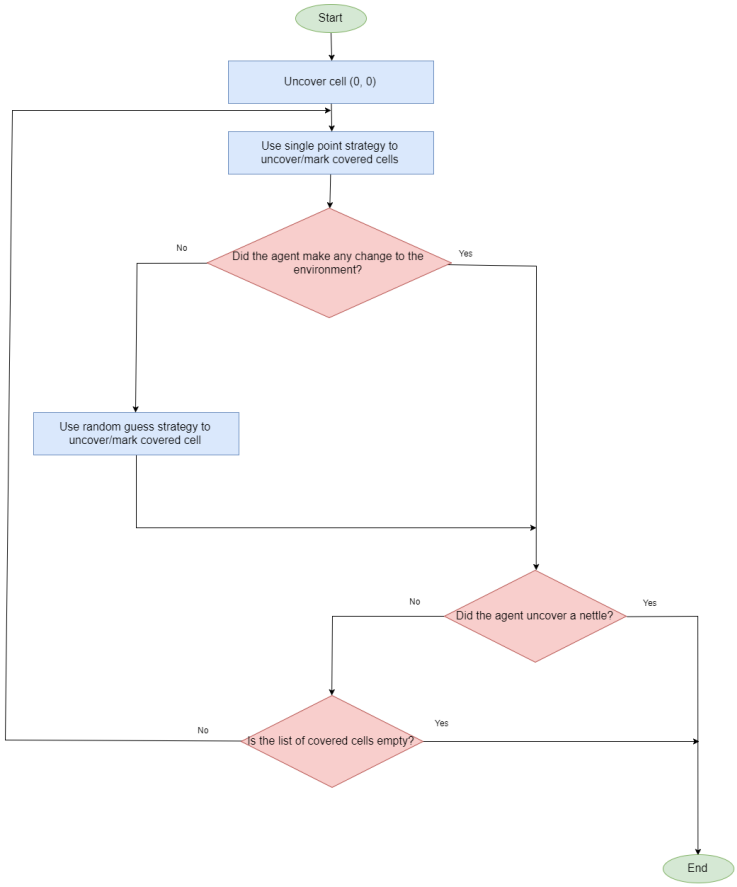


Figure 13: Flowchart of ***solveNettleWorld()*** method in ***BasicAgent*** class.

The agent used in part 1 is implemented in ***BasicAgent*** class. The main method used to solve the nettle world is ***solveNettleWorld()*** method. The flowchart showing the execution cycle of the method is shown in Figure 13.

```
protected void openCell(int row, int col) {
    System.out.println("reveal " + row + " " + col);
    // ask the game to reveal the number behind the cell
    int number = ns.getCellNumber(row, col);
    currentWorld[row][col].setNumber(number);
    // update list of uncovered and covered cells
    covered.remove(currentWorld[row][col]);
    uncovered.add(currentWorld[row][col]);
    if (number == 0) {
      openAllNeighborCells(row, col);
    }
    worldChanged = true;
    // check if the cell contain nettle
    gameOver = ns.isGameOver(currentWorld[row][col].getNumber());
  }
```

Listing 4: The code used to uncover a cell in nettle world.

As mentioned before, the actions that the agent can perform are uncover cell and mark cell as nettle. The code used to uncover a cell in the nettle world is shown in Listing 4. If the number behind the cell is 0 (no nettle around that cell), then the agent will uncover all neighbor cells. The code used to mark a cell as nettle is shown in Listing 5.

```
protected void markCell(int row, int col) {
    System.out.println("mark " + row + " " + col);
    // mark the cell indicating that it contains nettle
    currentWorld[row][col].setNumber(MARKED);
    // update list of uncovered and covered cells
    covered.remove(currentWorld[row][col]);
    marked.add(currentWorld[row][col]);
    worldChanged = true;
  }
```

Listing 5: The code used to mark a cell as nettle.

3.1.4 Random Guess Strategy

```
protected void randomGuessStrategy() {
    // pick a random cell from covered list and uncover that cell
    Random randomGenerator = new Random();
    int index = randomGenerator.nextInt(covered.size());
    Cell cell = covered.get(index);
    openCell(cell.getRow(), cell.getCol());
    randomGuess++;
  }
```

Listing 6: The code of RGS method.

Random guess strategy (RGS) picks a covered cell at random and uncovers it. The code of RGS method is shown in Listing 6.

3.1.5 Single Point Strategy

```
protected void singlePointStrategy() {
    for (int row = 0; row < currentWorld.length; row++) {
      for (int col = 0; col < currentWorld.length; col++) {
        if (covered.contains(currentWorld[row][col])) {
          takeAppropriateAction(row, col);
        }
      }
    }
  }
```

Listing 7: The code of SPS method.

```
private void takeAppropriateAction(int row, int col) {
    ArrayList<Cell> neighbors = getAllNeighbors(row, col);
    for (Cell neighbor : neighbors) {
      if (uncovered.contains(neighbor) || marked.contains(neighbor)) {
        if (allFreeNeighbors(neighbor)) {
          openCell(row, col);
          break;
        } else if (allMarkedNeighbors(neighbor)) {
          markCell(row, col);
          break;
        }
      }
    }
  }
```

Listing 8: The code of takeAppropriateAction() method.

Single point strategy (SPS) scans all covered cells one by one and take appropriate action based on information obtained from that cell (see Listing 7 and Listing 8). If a cell has All Free Neighbours (AFN), then the agent uncovers it. If a cell has All Marked Neighbours (AMN), then the agent marks it as nettle.

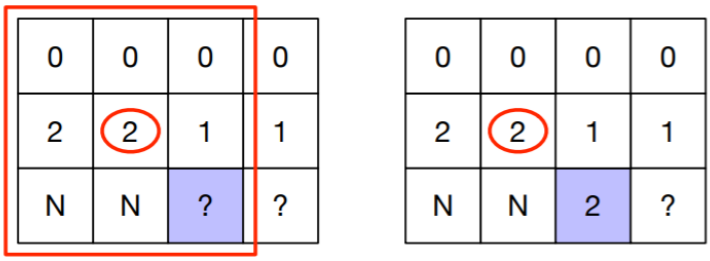


Figure 14: An example of all free neighbors situation (Toniolo 2017*a*).

AFN occurs when the effective number of an uncovered cell equals zero, i.e. the number equals the number of marked neighbors (see Figure 14) (Becerra 2015). In this situation, the agent can conclude that all adjacent unmarked cells are free cells. The code of AFN method can be found in Listing 9.

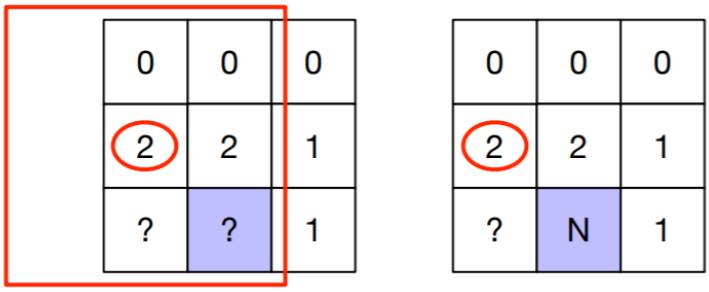


Figure 15: An example of all marked neighbors situation (Toniolo 2017*a*).

AMN occurs when the effective number equals the number of unmarked neighbors, all unmarked neighbors can be marked as nettles because the missing nettles cannot be in any other cells (see Figure 15) (Becerra 2015). The code of AMN method can be found in Listing 10.

```
private boolean allFreeNeighbors(Cell cell) {
    // true if: cellNumber == nettleCount
    ArrayList<Cell> neighbors = getAllNeighbors(cell.getRow(), cell.getCol()
   );
    int nettleCount = 0;
    for (Cell neighbor : neighbors) {
      if (neighbor.getNumber() == MARKED) {
        nettleCount++;
      }
    }
    if (cell.getNumber() == nettleCount) {
      return true;
    } else {
      return false;
    }
  }
```

Listing 9: The code of AFN method.

```java
private boolean allMarkedNeighbors(Cell cell) {
    // true if: cellNumber - nettleCount == unmarkedCount
    ArrayList<Cell> neighbors = getAllNeighbors(cell.getRow(), cell.getCol()
    );
    int nettleCount = 0;
    int unmarkedCount = 0;
    for (Cell neighbor : neighbors) {
      if (neighbor.getNumber() == UNMARKED) {
        unmarkedCount++;
      }
      if (neighbor.getNumber() == MARKED) {
        nettleCount++;
      }
    }
    if (cell.getNumber() - nettleCount == unmarkedCount) {
      return true;
    } else {
      return false;
    }
  }
```

Listing 10: The code of AMN method.

3.2 Examples and Testing

```
--------------------
0   0   0   2   #
0   0   0   2   #
1   2   1   2   #
#   #   #   #   #
#   #   #   #   #
--------------------
solving with SPS
mark 0 4
mark 1 4
reveal 2 4
reveal 3 3
reveal 3 4
reveal 4 4
reveal 4 3
mark 4 2
--------------------
0   0   0   2   F
0   0   0   2   F
1   2   1   2   1
#   #   #   2   0
#   #   F   2   0
--------------------
```

Figure 16: Printout from the program showing the action of the agent and the state of the game.

At each step, a statement is printed indicating the action of the agent and the state of the game (see Figure 16). For example, "reveal x y" for uncovering a cell in [x,y] coordinates, "mark x y" for marking the presence of a nettle in [x,y]. The program shows the state of the game before and after the agent's interaction. In this case, "#" indicates that the cell is covered, "F" indicates that the cell is marked as nettle, and the numbers indicates the number of nettles around the each cell. It also shows which strategy the agent is using to decide which cell to uncover or mark.

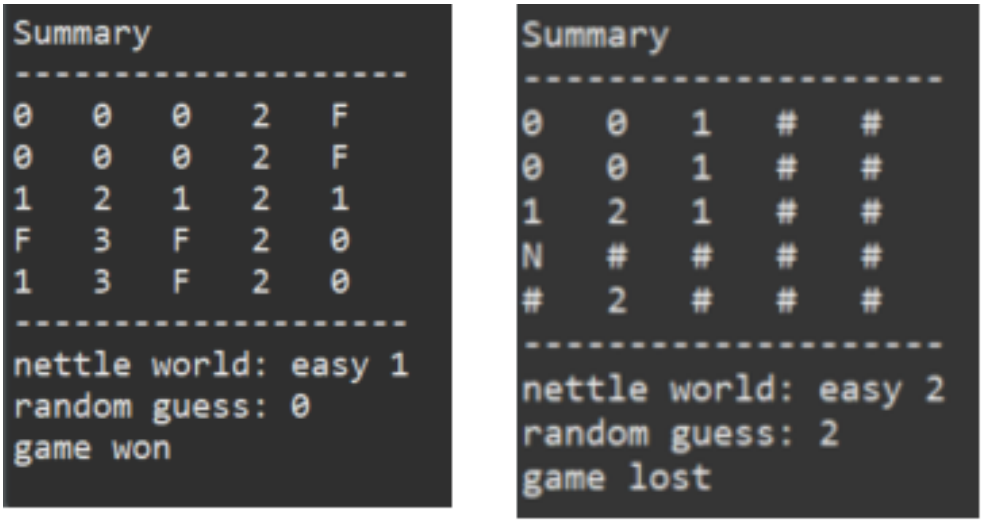


Figure 17: **Left:** summary printout when the game is won. **Right:** summary printout when the game is lost.

Figure 17 shows a summary printout of a game run. At the end of the game, the program shows the state of the nettle world and the number of random guess required to solve the nettle world. It also indicates if the game is won or lost.

World	Random guess (avg)	Success rate
Easy 1	0.0	100%
Easy 2	2.27	26%
Easy 3	1.12	44%
Easy 4	0.0	100%
Easy 5	1.53	47%

Table 1: The agent's performance in easy nettle worlds

World	Random guess (avg)	Success rate
Medium 1	0.0	100%
Medium 2	0.0	100%
Medium 3	5.68	25%
Medium 4	0.0	100%
Medium 5	1.57	47%

Table 2: The agent's performance in medium nettle worlds

World	Random guess (avg)	Success rate
Hard 1	2.46	42%
Hard 2	2.53	35%
Hard 3	1.51	53%
Hard 4	0.0	100%
Hard 5	0.0	100%

Table 3: The agent's performance in hard nettle worlds

The summary of the agent's performance in all nettle worlds can be found in Table 1, Table 2, and Table 3. The average number of random guess is obtained by taking the number of random guesses from 100 runs that the agent managed to solve the nettle world and calculate the average. The success rate is obtained by running the program 100 times and takes the number of times that the agent managed to solve the nettle world as success rate. The evaluation of the agent's performance can be found in Section 3.3.

3.3 Evaluation

Overall, the implementation of the agent managed to meet all of the requirements. The agent is capable of using single point reasoning strategy (SPS) and random guess strategy (RGS) to uncover/mark covered cells. It is clear that these two strategies are not enough to solve all of the nettle worlds provided. The success rate appears to be lower as the number of random guesses increase. The performance of the agent can be improved by implementing more reasoning strategies that would result in lower number of random guesses.

4 Part 2

4.1 Design

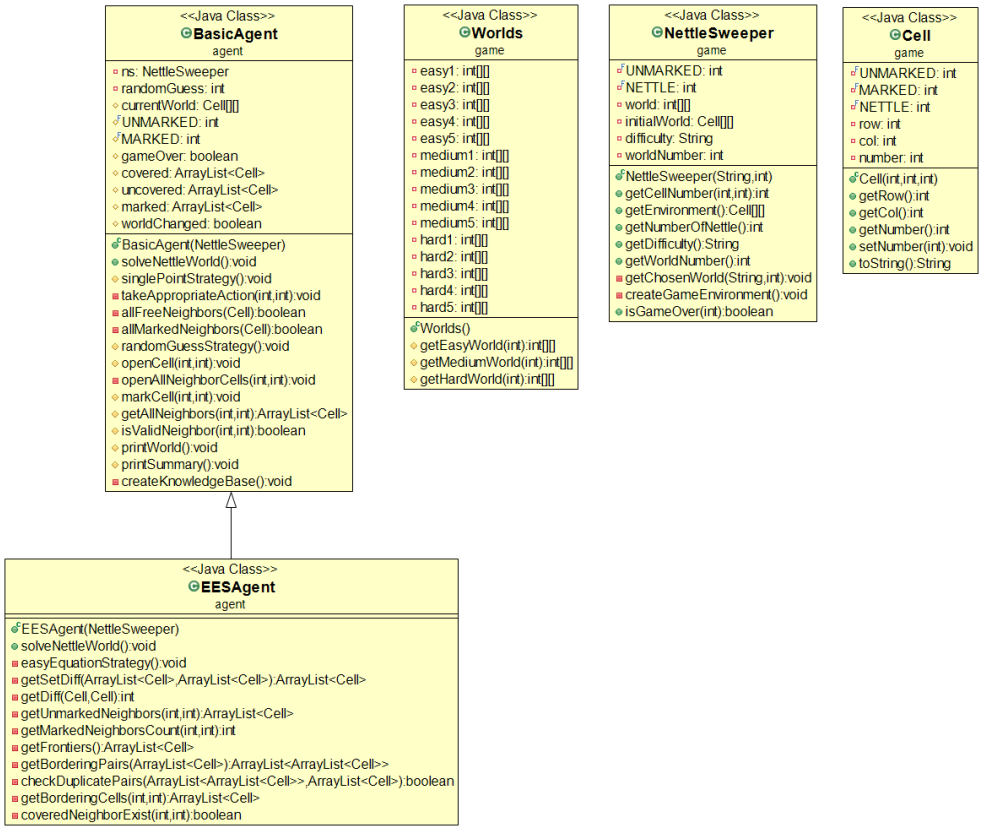


Figure 18: Class diagram for part 2.

4.1.1 Agent Implementation

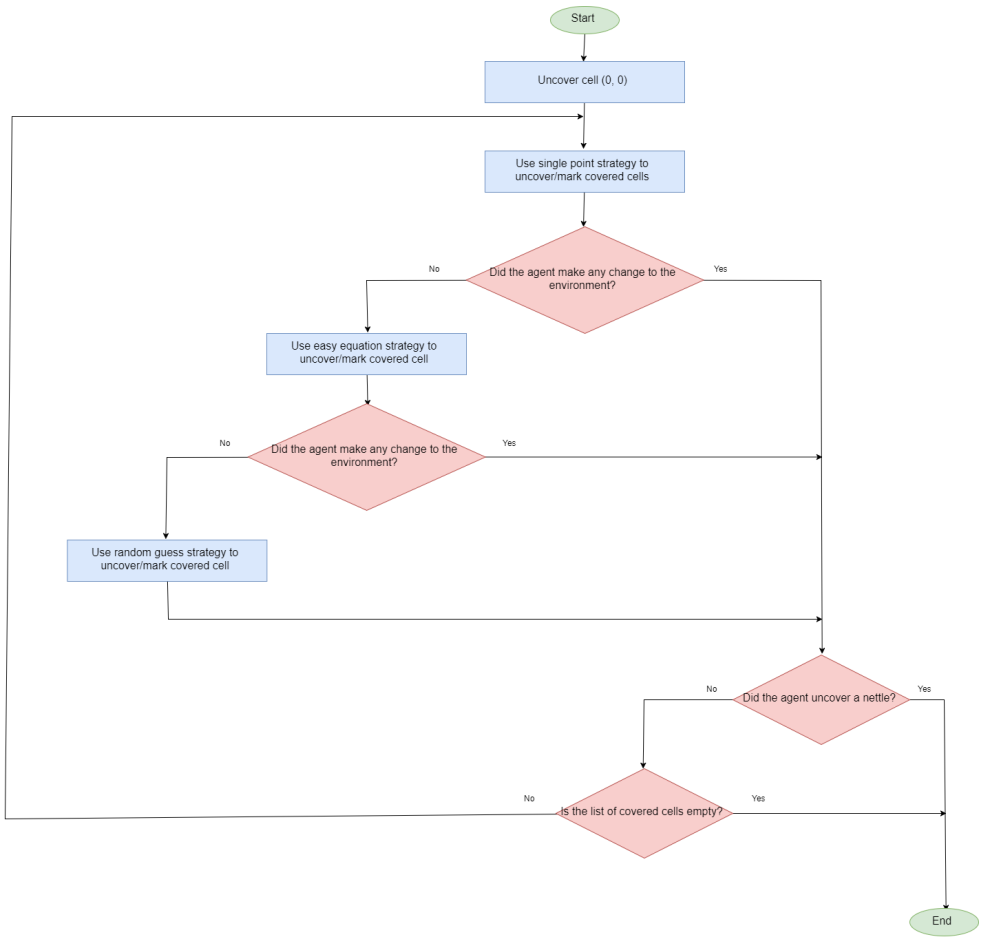


Figure 19: Flowchart of ***solveNettleWorld()*** method in ***EESAgent*** class.

The agent used in part 2 is implemented in ***EESAgent*** class. The class is extended from ***BasicAgent*** class, which means that it has access to SPS and RGS algorithms (see Figure 18). The main method used to solve the nettle world is ***solveNettleWorld()*** method. The flowchart showing the execution cycle of the method is shown in Figure 19.

4.1.2 Easy Equation Strategy

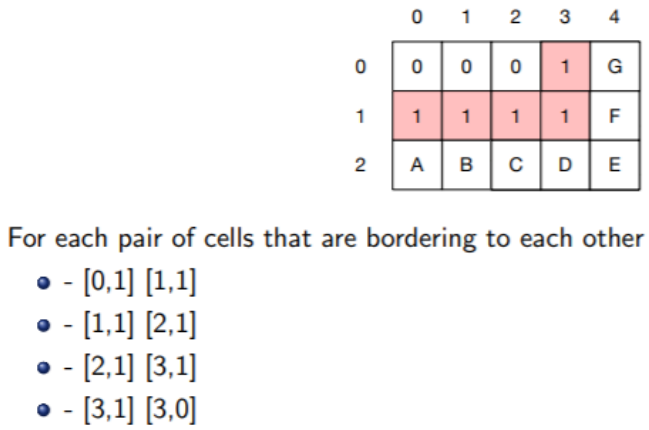


Figure 20: An example nettle world with frontiers highlighted in red (Toniolo 2017*a*).

Easy Equation Strategy (EES) gets all frontiers, i.e. cells which have at least one covered neighbor and creates a set of cell pairs that are bordering to each other (see Figure 20). For each pair [x,y][k,j], total difference in undiscovered nettle value can be calculated with the equation:

$$diff = |(v[x,y] - m[x,y]) - (v[k,j] - m[k,j])| \tag{1}$$

Where v[x,y] is the number of nettles contained in the cell and m[x,y] is the number of nettles already marked in the 8 neighbors of [x,y]. The next step is to find $S_{[x,y]}$ and $S_{[k,j]}$ (all uncovered/unmarked neighbors of [x, y] and [k, j]). If one set fully overlaps the other then continue, otherwise skip this pair. In the situation where one set fully overlaps the other, $S_{[k,j]} \setminus S_{[x,y]}$, the difference between $S_{[x,y]}$ and $S_{[k,j]}$ should be calculated. if the value of diff (calculated with Equation 1) is equal to 0, then uncover all cells in $S_{[k,j]} \setminus S_{[x,y]}$. If the value of diffis equal to $|S_{[k,j]} \setminus S_{[x,y]}|$, then mark all cells in $S_{[k,j]} \setminus S_{[x,y]}$. The code of ESS method can be found in Listing 11

```java
private void easyEquationStrategy() {
    ArrayList<Cell> frontiers = getFrontiers();
    ArrayList<ArrayList<Cell>> borderingPairs = getBorderingPairs(frontiers);
    Cell cellA, cellB;
    int diff;
    ArrayList<Cell> unmarkedSetA, unmarkedSetB, setDiff;
    for (int i = 0; i < borderingPairs.size(); i++) {
      // refer to the pair as cell A and cell B
      cellA = borderingPairs.get(i).get(0);
      cellB = borderingPairs.get(i).get(1);
      // calculate diff
      diff = getDiff(cellA, cellB);
      // get unmarked neighbors of both cells
      unmarkedSetA = getUnmarkedNeighbors(cellA.getRow(), cellA.getCol());
      unmarkedSetB = getUnmarkedNeighbors(cellB.getRow(), cellB.getCol());
      // check for overlaps
      // don't take any action if there is no overlap
      if (unmarkedSetA.containsAll(unmarkedSetB) || unmarkedSetB.containsAll(unmarkedSetA)) {
        setDiff = getSetDiff(unmarkedSetA,unmarkedSetB);
        // open cells if 0
        // mark cells if diff is equal to the size of setDiff
        // otherwise abandon
        if (diff == 0) {
          for (Cell cell : setDiff) {
            openCell(cell.getRow(), cell.getCol());
          }
        } else if (diff == setDiff.size()) {
          // need to fix marking logic
          for (Cell cell : setDiff) {
            markCell(cell.getRow(), cell.getCol());
          }
        }
      }
    }
  }
```

Listing 11: The code of ESS method.

4.2 Examples and Testing

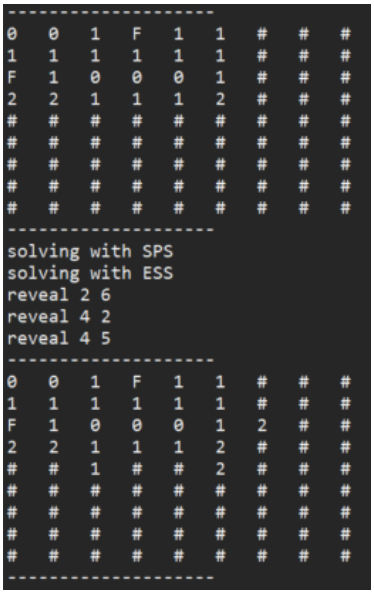


Figure 21: Printout from the program when the agent is using easy equation strategy to uncover/mark covered cells.

Like part 1, a statement is printed indicating the action of the agent and the state of the game. Figure 21 shows that the agent is using easy equation strategy to uncover/mark covered cells. An example game run summary is illustrated in Figure 22

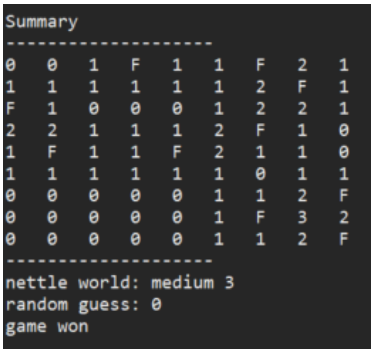


Figure 22: Summary printout when the agent managed to solve nettle world 3 on medium difficulty

World	Random guess (avg)	Success rate
Easy 1	0.0	100%
Easy 2	0.0	100%
Easy 3	0.0	100%
Easy 4	0.0	100%
Easy 5	0.0	100%

Table 4: The agent's performance in easy nettle worlds

World	Random guess (avg)	Success rate
Medium 1	0.0	100%
Medium 2	0.0	100%
Medium 3	0.0	100%
Medium 4	0.0	100%
Medium 5	0.0	100%

Table 5: The agent's performance in medium nettle worlds

World	Random guess (avg)	Success rate
Hard 1	0.0	100%
Hard 2	0.0	100%
Hard 3	0.0	100%
Hard 4	0.0	100%
Hard 5	0.0	100%

Table 6: The agent's performance in hard nettle worlds

The summary of the agent's performance in all nettle worlds can be found in Table

4, Table 5, and Table 6. The average number of random guesses and success rate are obtained with the same method as part 1 (see Section 3.2). The evaluation of the agent's performance can be found in Section 4.3.

4.3 Evaluation

For this part, the agent can use easy equation strategy (EES), single point strategy (SPS), and random guess strategy (RGS) to uncover/mark covered cells. It is clear that EES significantly improves the performance of the agent. When the agent cannot make further changes to the nettle world, it resorts to EES. The number of random guesses have been reduced to 0. The agent also have 100% success rate in all nettle worlds. However, there might be a board design that this agent cannot solve with current strategies. Due to time constraints, it is not possible to test the agent with more other board design.

5 Part 3

5.1 Design

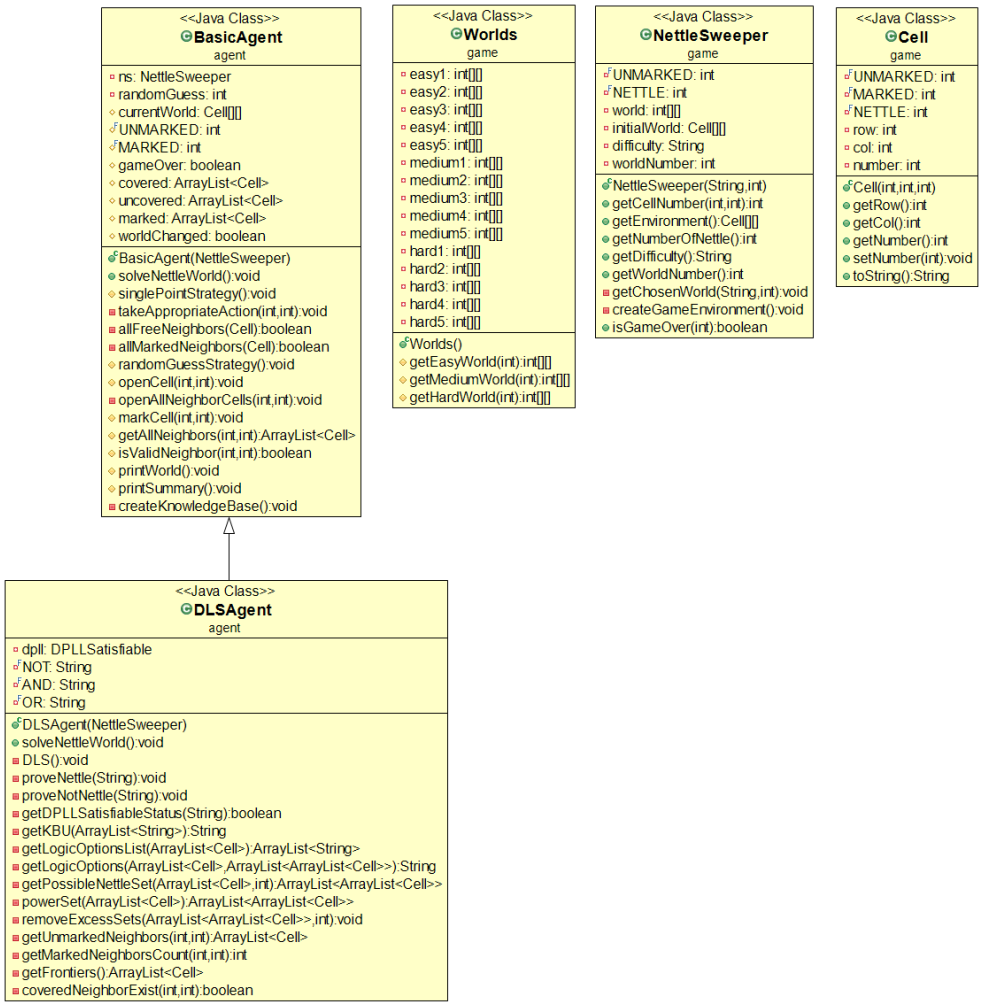


Figure 23: Class diagram for part 3.

5.1.1 Agent Implementation

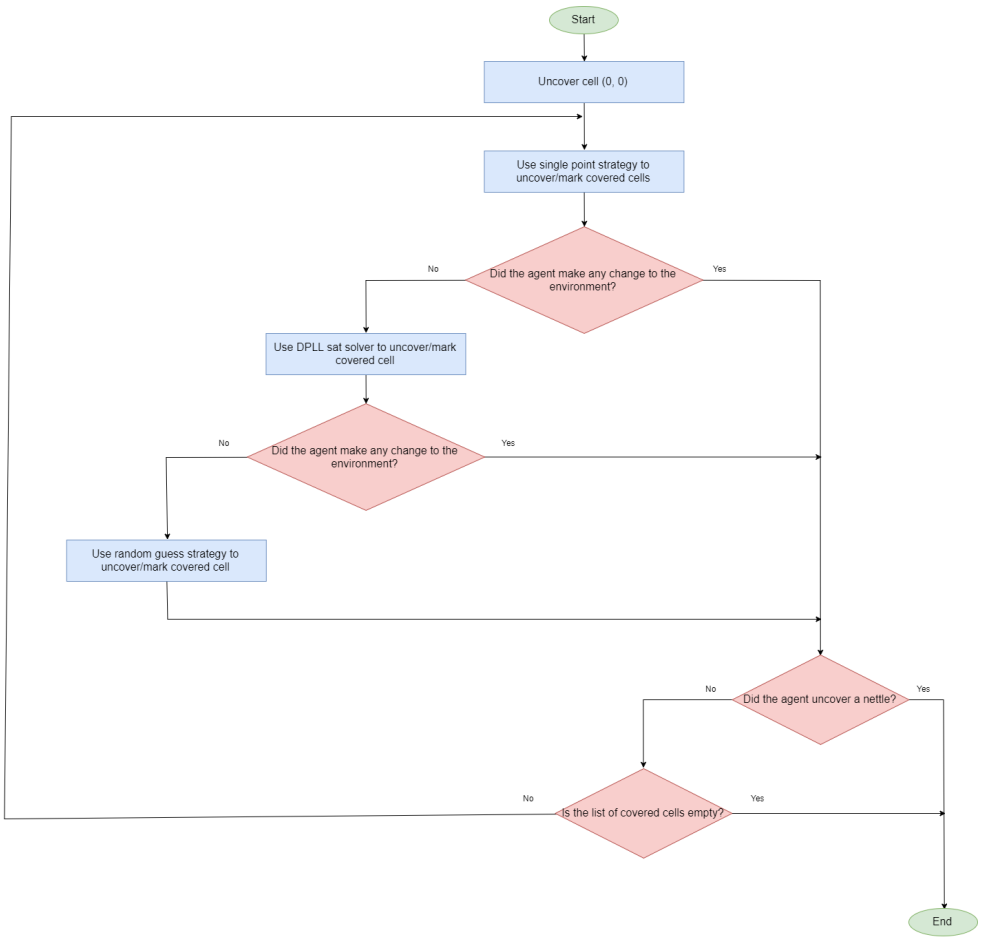
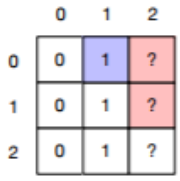


Figure 24: Flowchart of ***solveNettleWorld()*** method in ***DLSAgent*** class.

The agent used in part 3 is implemented in ***DLSAgent*** class. The class is extended from ***BasicAgent*** class, which means that it has access to SPS and RGS algorithms (see Figure 23). The main method used to solve the nettle world is ***solveNettleWorld()*** method. The flowchart showing the execution cycle of the method is shown in Figure 24.

5.1.2 DPLL Sat Solver



- $(N_{2,0} \wedge \neg N_{2,1}) \vee$
  $(\neg N_{2,0} \wedge N_{2,1})$

Figure 25: Options in logic (Toniolo 2017*b*).

```
private String getLogicOptions(ArrayList<Cell> unmarked
      , ArrayList<ArrayList<Cell>> possibleNettleSet) {
    String options = "(";
    for (int i = 0; i < possibleNettleSet.size(); i++) {
      if (i != 0) {
        options += " " + OR + " ";
      }
      options += "(";
      for (int j = 0; j < unmarked.size(); j++) {
        Cell cell = unmarked.get(j);
        if (j != 0) {
          options += " " + AND + " ";
        }
        if (!possibleNettleSet.get(i).contains(cell)) {
          options += NOT;
        }
        options += "N" + cell.getRow() + cell.getCol();
      }
      options += ")";
    }
    options += ")";
    return options;
  }
```

Listing 12: The code used to get logic options.

Similar to EES, the first step is to get all the frontiers. Then, propositional logic is used to represent the facts about each cell in the frontiers and its neighbors (see Figure 25). In this case, N indicates that there is a nettle in that cell. The code used to obtain logic options can be found in Listing 12. Now that the agent has a full representation of the nettle world for those cells uncovered. This complex proposition is called KBU. An example of a KBU is shown in Figure 26.

$$KBU = [(N_{2,0} \wedge \neg N_{2,1} \wedge \neg N_{2,2}) \vee (\neg N_{2,0} \wedge N_{2,1} \wedge \neg N_{2,2}) \vee (\neg N_{2,0} \wedge \neg N_{2,1} \wedge N_{2,2})] \wedge [(N_{2,0} \wedge \neg N_{2,1}) \vee (\neg N_{2,0} \wedge N_{2,1})] \wedge [(N_{2,1} \wedge \neg N_{2,2}) \vee (\neg N_{2,1} \wedge N_{2,2})]$$

Figure 26: An example of KBU in nettle world (Toniolo 2017*b*).

To prove that there is a nettle in cell [x, y], DPLL satisfiable status of $KBU \wedge \neg N_{x,y}$ needs to return false. To prove that cell [x, y] is clear, DPLL satisfiable status of $KBU \wedge N_{x,y}$ needs to return false. The code of DLS method can be found in Listing 13.

```
private void DLS() {
    ArrayList<Cell> frontiers = getFrontiers();
    // create options for all cells
    ArrayList<String> optionsList = getLogicOptionsList(frontiers);
    // create KBU string
    String KBU = getKBU(optionsList);
    // try to find nettles first
    proveNettle(KBU);
    proveNotNettle(KBU);
  }
```

Listing 13: The code of DLS method.

5.2 Examples and Testing

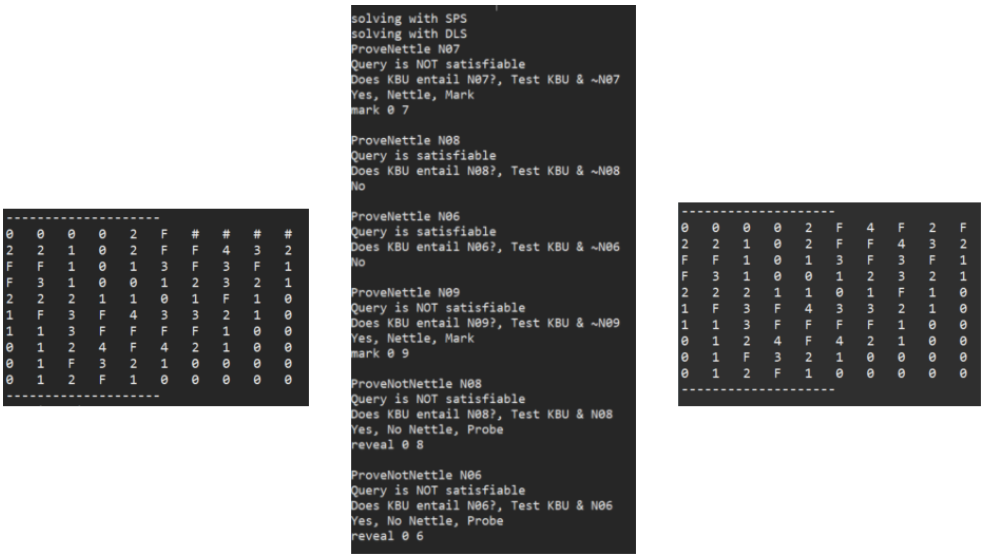


Figure 27: **Left:** state of the board before agent's interaction. **Middle:** printout from the program when the agent is using satisfiability information obtained with a DPLL solver to uncover/mark covered cells. **Right:** state of the board after agent's interaction.

Figure 27 shows that the agent is using information obtained with a DPLL solver to uncover/mark covered cells. An example game run summary is illustrated in Figure 28.

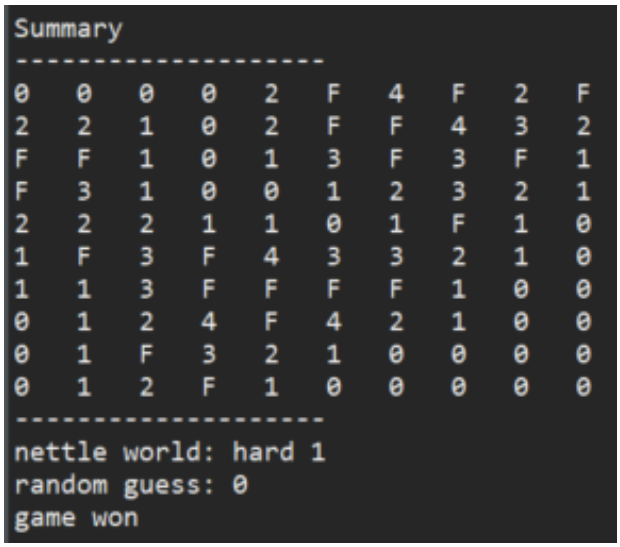


Figure 28: Summary printout when the agent managed to solve nettle world 1 on hard difficulty

World	Random guess (avg)	Success rate
Easy 1	0.0	100%
Easy 2	0.0	100%
Easy 3	0.0	100%
Easy 4	0.0	100%
Easy 5	0.0	100%

Table 7: The agent's performance in easy nettle worlds

World	Random guess (avg)	Success rate
Medium 1	0.0	100%
Medium 2	0.0	100%
Medium 3	Unsolvable	Unsolvable
Medium 4	0.0	100%
Medium 5	0.0	100%

Table 8: The agent's performance in medium nettle worlds

World	Random guess (avg)	Success rate
Hard 1	0.0	100%
Hard 2	0.0	100%
Hard 3	0.0	100%
Hard 4	0.0	100%
Hard 5	0.0	100%

Table 9: The agent's performance in hard nettle worlds

The summary of the agent's performance in all nettle worlds can be found in Table 7, Table 8, and Table 9. The average number of random guesses and success rate are obtained with the same method as part 1 (see Section 3.2). The evaluation of the agent's performance can be found in Section 5.3.

5.3 Evaluation

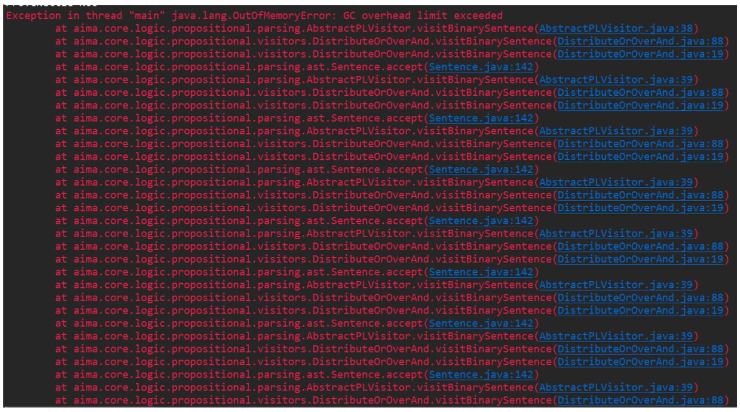


Figure 29: Error that appears when trying to solve nettle world 3 on medium difficulty

```
—————————————————————————————————
0  0  1  F  1  1  #  #  #
1  1  1  1  1  1  #  #  #
F  1  0  0  0  1  #  #  #
2  2  1  1  1  2  #  #  #
#  #  #  #  #  #  #  #  #
#  #  #  #  #  #  #  #  #
#  #  #  #  #  #  #  #  #
#  #  #  #  #  #  #  #  #
#  #  #  #  #  #  #  #  #
—————————————————————————————————


KBU = ((~N06 & N16) | (N06 & ~N16)) & ((~N16 & ~N06 & N26) | (~N16 & N06 & ~
    N26) | (N16 & ~N06 & ~N26)) & ((~N26 & ~N16 & N36) | (~N26 & N16 & ~N36)
    | (N26 & ~N16 & ~N36)) & ((~N40 & N41) | (N40 & ~N41)) & ((~N41 & ~N42 &
    N40) | (~N41 & N42 & ~N40) | (N41 & ~N42 & ~N40)) & ((~N42 & ~N43 & N41)
    | (~N42 & N43 & ~N41) | (N42 & ~N43 & ~N41)) & ((~N43 & ~N44 & N42) | (~
    N43 & N44 & ~N42) | (N43 & ~N44 & ~N42)) & ((~N44 & ~N45 & N43) | (~N44 &
     N45 & ~N43) | (N44 & ~N45 & ~N43)) & ((~N45 & ~N36 & ~N26 & N46 & N44) |
     (~N45 & ~N36 & N26 & ~N46 & N44) | (~N45 & N36 & ~N26 & ~N46 & N44) | (
    N45 & ~N36 & ~N26 & ~N46 & N44) | (~N45 & ~N36 & N26 & N46 & ~N44) | (~
    N45 & N36 & ~N26 & N46 & ~N44) | (N45 & ~N36 & ~N26 & N46 & ~N44) | (~N45
     & N36 & N26 & ~N46 & ~N44) | (N45 & ~N36 & N26 & ~N46 & ~N44) | (N45 &
    N36 & ~N26 & ~N46 & ~N44))
```

Listing 14: The state of the game before the error occurs.

For this part, the agent is able to use DPLL solver (DLS), single point strategy (SPS), and random guess strategy (RGS) to uncover/mark covered cells. It is clear that DLS significantly improves the performance of the agent when compared to part 1. When the agent cannot make further changes to the nettle world, it uses information obtained with a DPLL solver to uncover/mark covered cells. The number of random guesses have been reduced to 0. The agent also have 100% success rate in all nettle worlds except nettle

world 3 on medium difficulty. The error that appears when trying to solve nettle world 3 on medium difficulty is shown in Figure 29. The state of the game before the error occurs can be found in Listing 14. This might be an issue that relates to the implementation of DPLL algorithm. Due to time constraints, it is not possible to identify the cause of this error.

6 Running

The program in part 1, part 2, and part 3 take difficulty and nettle world number as parameters. Difficulty parameters are listed below.

- easy
- medium
- hard

Nettle world numbers are numbers in range [1,6].

6.1 Logic 1

Logic1.jar can be executed with the command:

```
java -jar Logic1.jar difficulty world_number
```

Listing 15: The command used to run Logic1.jar.

To run Logic1.jar from NettleSweeper folder with nettle world 1 on easy difficulty, the command in Listing 16 should be used.

```
java -jar Logic1.jar easy 1
```

Listing 16: The command used to run Logic1.jar from NettleSweeper folder.

6.2 Logic 2

Logic2.jar can be executed with the command:

```
java -jar Logic2.jar difficulty world_number
```

Listing 17: The command used to run Logic2.jar.

To run Logic2.jar from NettleSweeper folder with nettle world 1 on medium difficulty, the command in Listing 18 should be used.

```
java -jar Logic2.jar medium 1
```

Listing 18: The command used to run Logic2.jar from NettleSweeper folder.

6.3 Logic 3

Logic3.jar can be executed with the command:

```
java -jar Logic3.jar difficulty world_number
```

Listing 19: The command used to run Logic3.jar.

To run Logic3.jar from NettleSweeper folder with nettle world 1 on hard difficulty, the command in Listing 20 should be used.

```
java -jar Logic3.jar hard 1
```

Listing 20: The command used to run Logic3.jar from NettleSweeper folder.

Bibliography

aimacode (2017), 'Aima3e-java (jdk 8+)', https://github.com/aimacode/aima-java. Accessed: 2017-11-19.

Becerra, D. J. (2015), Algorithmic Approaches to Playing Minesweeper, PhD thesis.

Donner, R. & Johnson, C. (2013), 'Windows minesweeper', http://www.minesweeper.info/wiki/Windows_Minesweeper. Accessed: 2017-11-16.

Toniolo, A. (2017*a*), 'Cs5011 logical agents - nettle sweeper strategies', https://studres.cs.st-andrews.ac.uk/CS5011/Lectures/L13_w8.pdf. Accessed: 2017-11-20.

Toniolo, A. (2017*b*), 'Cs5011 logical agents - nettle sweeper strategies', https://studres.cs.st-andrews.ac.uk/CS5011/Lectures/L14_w8.pdf. Accessed: 2017-11-20.

Toniolo, A. (2017*c*), 'Cs5011 logical agents recap and assignment intro', https://studres.cs.st-andrews.ac.uk/CS5011/Lectures/L12_w7.pdf. Accessed: 2017-11-19.