CS5011 P2

160004864

1 November 2019

1 Introduction

For this practical we were required to implement a logical agent with the ability to play and solve the Tornado Sweeper game. In a similar fashion to the Minesweeper game, the goal of the game is to uncover all the cells on a hexagonal board but those containing a tornado. In the scenario in which the agent uncovers a cell containing a tornado, the game ends and the agent loses. The specification outlines several rules of Tornado Sweeper, which the program must follow and it describes different strategies that the agent can adapt in order to play the game such as the 'Single Point Strategy for Hexagonal Worlds' (SPX) and the 'Satisfiability Strategy for Hexagonal Worlds' (SATX).

1.1 Implementation Checklist

- Implementation of the basic agent using a random probing strategy (RPX). In addition, the agent is able to establish whether the game is over and it also prints the required statements describing the actions it is taking along with the state of the agent's view of the board.
- Implementation of the intermediate agent with the ability to use:
    1. The SPX strategy in order to determine whether to flag or probe the next cell depending on whether the cell is in an 'All Marked Neighbours' (AMN) or an 'All Free Neighbours' situation.
    2. The SATX strategy which involves translating the knowledge base into a logic sentence and then using satisfiability results from a SAT solver in order to select the next move.
- **Extension** Implementation of a program which allows for the comparison of the three strategies (RPX, SPX and SATX) to be made. This is accomplished by evaluating the agent's performance after playing in all the 30 'worlds' for a specified number of times and keeping track how many times each agent won.
- **Extension** Researched and reported findings regarding additional strategies to improve the performance of the logic agent.

1.2 Compiling and Running Instructions

First navigate to the base directory, **P2**. Then in order to run the program with the already compiled source code (found in the **P2/out/production/P2** folder , you may use the following command:

```
java -cp sat4j.jar:antlr.jar:logicng.jar:out/production/P2 A2main
     <RPX|SPX|SATX> <ID>
```

For example, to run the agent using the SPX strategy on world M5, use the following command:

```
java -cp sat4j.jar:antlr.jar:logicng.jar:out/production/P2 A2main
     SPX M5
```

In order to re-compile the source code, you may use the following command:

```
javac -cp sat4j.jar:antlr.jar:logicng.jar -d out/production/P2
    A2src/*.java
```

In order to run the program comparing the performance of the agents, run the following command:

```
java -cp sat4j.jar:antlr.jar:logicng.jar:out/production/P2 A2Test
     [iterations]
```

The iterations parameter is optional and if not included, the program will carry out 5 iterations over the worlds by default. This program will play the game with all the agents, for all the worlds for a `[iterations]` number of times, and in the end it will print the results.
Caution: When copying and pasting the above into the terminal, make sure there aren't any excess or omitted spaces as a result of the line breaks.
Note: I've included the 'non standard libraries' in the **A2src/** folder as well, but it is recommended to run the program as described above, using the libraries found in the base directory.

2 Design, Implementation and Evaluation

2.1 PEAS Model

- Performance measure: The primary performance measure is whether the agent has won the game or not. Another, more sophisticated performance measure can be the number of tornadoes left on the board compared to the total number of covered cells, expressed as a percentage, with 100% meaning the agent has won the game as all the cells that are covered, are tornadoes.

- Environment: The environment is the rhombus shaped Hexagonal grid with N cells per side.

- Actuators: The logical agent, which chooses to either probe or mark a cell depending on the strategy being used.
- Sensors: When probed, if a cell is clear it will give a hint on the number of adjacent cells that contain a tornado. If a cell contains a tornado, the agent will be able to perceive that and the game will be over.

2.2 Game infrastructure

The Game infrastructure is implemented in the Game.java file, which contains a class definition representing an instance of a Game. As required, this class has the following features:

- It holds the actual view of the Tornado Sweeper world in an attribute of type `Board`. The underlying data structure of this type is a two dimensional array of `char`.
- It is able to tell the agent about perceptions. This is implemented in the `uncoverCell` method which takes two `int` parameters representing the x and y coordinates of a cell to be uncovered and it returns a Cell object which contains the percept. The percept will either be a hint on the number of tornadoes surrounding the cell or an indication that a tornado has been found.
- It can determine whether a game is over and whether the game has be won or lost. The Game instance has `boolean` attributes `gameOver` and `gameWon` which hold whether the game has ended and whether the game has been won. The `gameOver` is set to true if a cell uncovered is a tornado or when a game has been won which is also when the `gameWon` is set to true. The `checkGameWon` is the method which checks whether the game has been won. It can determine whether the game has been won by checking that all the cells that have yet to be uncovered (held in the `coveredCells` list) are cells containing a tornado.
- It is independent from the agent. Independence is achieved by keeping the two implementations separate and have each of them keep their own view of the board. The Game instance holds a complete view of the board (knows what each cell contains at all times) while the Agent instance holds a partial view of the board and it has information only on the cells is has probed. The only piece of information the Agent receives from the actual board view is its length. Throughout the game the Agent communicates with the Game instance through the methods described above.

2.3 Agent Infrastructure

The Agent infrastructure is implemented in the Agent.java file which contains a class definition representing an instance of an Agent. As required, this class has the following features:

- It holds a knowledge base indicating the cells that are yet to be probed, information about the cells already uncovered as well as storing cells that are marked as tornadoes. The Agent holds all this information in several lists that it has as attributes. The `unexaminedCells` list holds the cells that are 'unknown cells' that is cells that haven't been probed or marked. The `examinedCells` list holds the cells that have either been probed or marked as a danger. The `uncoveredCells` list holds the cells that have been probed. Furthermore, the `tornadoCells` list holds the cells that have been marked as a danger and finally the `allCells` array holds all the cells. In the beginning all the Cell objects have their `hint` attribute as '?' indicating they are unexamined. Throughout the game, the cells have their hint changed depending on the precepts received from the Game instance and they're added/removed to and from the appropriate lists.

- Actions such as probing a cell and marking a cell. The action of probing a cell is implemented in the `probeCell` which takes a Cell object, representing the cell to be probed, as a parameter. In this method, the Game's `uncoverCell` method is called to get the percept of that cell. The cell's hint attribute is then changed to the perceived hint and the cell is then removed from the `unexaminedCells` list and it is added to the `examinedCells` and the `uncoveredCells` lists. The action of marking a cell is implemented in the `markCell` which takes a Cell object, representing the cell to be marked, as a parameter. In this method, the cell's hint is set to 'D', indicating it is a marked danger, and it is added to the `tornadoCells` and `examinedCells` lists and removed from the `unexaminedCells` list.

- Strategies to probe the next cell. The strategies implemented are RPX, SPX and SATX and they will be described in the next section.

2.4 Agent Strategies

Note I: Whichever strategy is being used, the agent will always use the `clearNeighbours` method which probes all the adjacent cells of a cell whose hint is 0 i.e. there are no tornadoes around it.
Note II: Due to the word count limit, I don't go into much depth regarding the implementation of each individual strategy in this section. The reader is encouraged to check the included, heavily commented source code for more details.

2.4.1 RPX

This is a pretty straightforward strategy as it does not involve any use of logic and it is implemented in the `makeRandomMove` method. Rather than using logic, the `Random` class is used to generate a random value between 0 and (not including) the size of `unexaminedCells`. This value is then used to get a Cell object from the aforementioned list, in order to be probed.

2.4.2 SPX

The agent uses this strategy, implemented in the `makeSPXMove`, in order to flag or probe the next cell. The method first iterates over the cells in the `unexaminedCells`. Then for each of these cells, its neighbours are returned by the `getAllNeighbours` and the agent checks whether the cells is in an All Free Neighbours (AFN) or All Marked Neighbours (AMN) situation. As described in the lectures[1], in the situation of AFN, the cell under consideration can be safely probed and in the situation of AMN, the cell under consideration has to be marked as a danger, as it will contain a tornado. If none of the cells are found in an AFN or AMN situation the agent makes a random move instead in order to procede with the game.

2.4.3 SATX

This strategy is implemented in the `makeSATMove` method and it is used to determine whether a given cell does or does not contain a tornado. This is achieved by taking the following steps:

- The agent's knowledge base is translated into a logical sentence by the `translateKB` and the `createClause` methods. These methods iterate the `uncoveredCells` list and by examining each cell's surroundings, a `String` holding logical sentence containing all the possibilities of where tornadoes could be is created. This sentence it is then turned into a `Formula` object using the logicng library[2] along with the antlr library[3].
- Since the SAT4J Core solver[4] needs the input in a CNF DIMACS format it becomes necessary to translate the aforementioned logical formula. The DIMACSGenerator.java file contains a class implementation that has the ability to carry out the translation. First, upon receiving the formula, the `convertToDIMACS` method turns the formula into CNF as required, and all the literals are encoded to `int` and the mapping between the two is preserved in the `literalsHashMap` attribute of the class. It is worthwhile to note that the negation of each literal found in the formula is also encoded, so that in later attempts to use the SAT solver for proof by contradiction, the clause being checked is already encoded. Then, for each clause of the CNF formula an array of `int` containing the encoded value of the literals is returned by the `getClause` method. Finally the `convertToDIMACS` will return a 2-d `int` array containing the encoded, CNF version of the knowledge base.
- After completing the above two steps, we're in a position to use the SAT solver to test whether is it possible for a cell to contain a tornado or not. This is done by checking whether the negation of the clause indicating that a cell is found in that cell is satisfiable or not. It it is unsatisfiable then it means that it is not possible for that cell to contain a tornado and thus it can be safely probed. If it is satisfiable then that cell can possibly contain a tornado and thus it would be unsafe to probe it. If the agent

cannot determine that any of the covered cells can be safely probed, the agent will make a random move to proceed with the game.

2.5 Agent Performance Evaluation

As described in section 1.1, I've created a program which lets the agent play on all 30 'worlds' with each of the three strategies for a specified number of times. The following are the results after 100 iterations (i.e. each agent strategy playing each of the 30 worlds for 100 times).

Strategy	**Wins**	**Win %**
RPX	22/3000	1%
SPX	1872/3000	62%
SATX	2594/3000	86%

It is fairly obvious that the strategies listed in order of decreasing performance are SATX, SPX, RPX. It is trivial as to why RPX is the worst performer of the group but it is worth reasoning why SATX is superior to SPX. Since the SAT strategy utilises information taken from the entire knowledge base it can make better decisions compared to the SPX strategy which only focuses on a single point before making a decision. By having more information and a more complete picture of the agent's view of the board, the SATX strategy achieves superior performance. It is worth mentioning that the success of each strategy indicates the amount of (or lack of) random probing required. Since agents using the SPX or SATX strategy will never probe a cell unless they can logically entail that it does not contain a tornado, we can infer that all the losses are caused by moves made at random. Indeed we can see that this is true by examining the SPX_all.txt and SATX_all.txt files. As a result, we can see that SATX is less likely to require a random move compared to SPX. A screenshot of the final output of the A2Test program is included in the submission with the file name 'Results.png'.

2.6 SPX and SATX combination

It is easy to see that if an agent using the SATX strategy cannot determine whether a cell can be safely probed then neither can an agent using the SPX strategy. At the same time, if an SPX strategy agent had the ability to use a SAT solver to determine whether a cell contains a tornado, then it would be just as powerful as the agent using the SATX strategy.

3 Improving the performance of the logic agent

By observing that the agents using the SPX and SATX strategies lose because of 'bad' random choices, it becomes obvious that making this random choice more informed becomes a priority in order to improve an agent's performance. Some interesting suggestions are made in the Minesweeper wiki [5].

- When facing a random choice, prefer to choose a corner cell. This is usually the better choice and is even the suggested opening move in advanced Minesweeper tactics as the corners have the biggest chance of an opening i.e. a state of the board which allows a logical agent to carry on.
- Not all random choices are equal. When the agent's strategy cannot determine the next move, rather than making a random choice, an agent could produce all the possible mine configurations around probed cells and calculate the probabilities of a mine being in each of the covered cells and choose the one with the lowest probability. Obviously, one may also choose a covered cells that is not adjacent to a probed cell and hope for the best. This probability can also be calculated by considering the number of cells probed *nProbed* , the number of cells adjacent to probed cells *nAdjacent*, the total number of cells on the board *nCells*, the number of possible mines around probed cells (may use maximum value) *nMinesAdjacent* and the number of mines that have not been marked yet *nMines.* This approach is demonstrated with an example here[6], and the probability can be calculated using the formula:

$$\frac{nMines - nMinesAdjacent}{nCells - (nAdjacent + nProbed)} \tag{1}$$

- Be efficient. When coming across a forced random choice i.e. a choice between two cells on an edge, it's a waste of time to move on to another part of the board and keep solving the puzzle as eventually the random choice will have to be made. Thus whenever faced with a situation where a random choice has to be made, do it as soon as possible, allowing an agent to move on with the next game if they lose.

4 Testing

The aspects/features of the program whose correctness need to be tested, can be separated into two distinct categories.

- Game correctness - we must test that the game is being played correctly, following all of the rules.
- AI logic correctness - we must test that the AI strategies are working the way they're supposed to.

Note: I will be referring to text files included in the submission to demonstrate the correctness of the program. These tests are run on the Small world S1, to keep the files short and succint.

4.1 Game Correctness

- 'Hint cells' are probed first: in the RPX_S1.txt file, we can see that at t1, the middle and top left corner are probed first.

- The cells adjacent to a probed cell with a hint value of 0 are also probed: In the same file, we can see that once the top left corner cell which contains the number 0 is probed, all the adjacent cells are also probed as seen in the output, under the message 'Uncovering free neighbour'.
- The game ends when a cell containing a tornado is found: As seen at the end of the file, the cell at 3,4 is probed. This cell contains a tornado and the game is over and the appropriate 'game lost' message is displayed.
- The agent may use flags to signal the presence of a Tornado in covered cells: In the SPX_S1.txt file we can see the output of the agent playing the S1 world using the SPX strategy. As we can see, the cells 4,0 3,3 3,4 1,4 and 0,4 are correctly marked using the letter 'D' for danger.
- The agent wins the game if all but T cells are probed: At the end of the same file, we can see that all the cells apart from the ones marked have been probed and that the appropriate 'game won' message is displayed indicating that it works as intended.

4.2 Agent Logic Correctness

- RPX correctness: The correctness of the logical agent using RPX can be proved trivially, by comparing the output produced by the agent using this strategy on the same world in two different runs. i.e. each time the sequence of cells chosen to be probed are different. As we can see in RPX_S1.txt file, after uncovering all the free neighbours, the agent chooses to probe 3,4 which contains a tornado and ending the game, while in RPX_S1_2.txt file after uncovering all the free neighbours, the agent probes 2,4 and then 4,4 to win the game. Clearly they've made different choices and thus we can infer that it does indeed choose the next cell to be probe randomly. The output produced by the agent using the RPX strategy in all the worlds, can be found in the RPX_all.txt file.
- SPX correctness: In order to prove that this strategy is correctly implemented by the agent we need to establish that the only time a cell containing a tornado is probed is when no cell is in an AFN or AMN situation and is the result of the agent making a move at random. This can be verified by looking at the output produced by the agent using the SPX strategy in all of the worlds, found in the SPX_all.txt file. Indeed in all the cases the agent lost, the move that caused it was one that was randomly made. Furthermore it's easy to examine that in all of such cases there was no cell on the board in an AMN of AFN situation. Thus, we can see that the SPX strategy is correctly implemented.
- SATX correctness: In a similar fashion to the above, the correctness of this strategy is established by showing that whenever the agent using the strategy loses, it's the result of making a random move. This happens when the SAT solver cannot determine that any of the covered cells can

be safely probed, given the current knowledge base.The correctness can be verified by looking at the output produced by the agent using the SATX strategy in all of the worlds, found in the SATX_all.txt file. Indeed, we can see that whenever the agent using this strategy has lost the game, it was because it had to make a random move to proceed.

5 Conclusion

Overall, I'm pleased with the result of my work as I have managed to successfully implement the agent using the three strategies, RPX, SPX and SATX. In addition, I have managed to implement a program that is very useful in comparing the performance of these strategies, allowing me to determine which approach is superior and examine the reasons why that is. Lastly, I have included a section in this report containing several suggestions about ways that could improve the performance of the logical agent. The part that I found most difficult was translating the knowledge base into a logical sentence and then produce the CNF DIMACS encodings to be used by the SAT4J solver and making sure that was done correctly. If I had more time I would attempt extending the program to permit the agent to play on boards of a different shape and I would also use the knowledge I've gained using the satisfiability approach to solve the Tornado Sweeper problem, to solve other games such as Sudoku. This assignment was very interesting as it used a variant of a familiar game, Minesweeper, in order to give me an insight into different strategies that can be used in order to solve logic games and other logic problems in AI.

References

[1] Toniolo, A. (2019). CS5011(L8 W4): CS5011 Logical Agents - The Danger Sweeper

[2] LogicNG16 for Java https://github.com/logic-ng/LogicNG16

[3] ANTLR https://www.antlr.org/

[4] SAT4J Core https://www.sat4j.org/products.php

[5] http://www.minesweeper.info/wiki/Strategy#Patterns

[6] https://puzzling.stackexchange.com/questions/50948/optimal-next-move-in-minesweeper-game