Logic Tornado Sweeper Report

1 Introduction

1.1 Meta Information

Assignment: Practical 3

Submission Date: March 27th, 2023

ID: 220011642

1.2 Checklist of Implemented Parts

· Part 1: Completed, fully working

· Part 2: Completed, fully working

· Part 3: Completed, fully working

· Part 4: Completed, fully working

· Part 5: Completed, fully working

1.3 Compiling and Running Instructions

The execution of the program is designed as the spec required.

To run the code, simply navigate under /src and use this command:

```
./playSweeper.sh <Pn> <World ID> [verbose]
```

For example, to run Part 2 on Test1 where each step is displayed, one can type:

```
./playSweeper.sh P2 TEST1 verbose
```

To run Part 4 on World SMALL5, one can type:

```
./playSweeper.sh P4 SMALL5
```

To run Part 5 on World REVTEST1, one can type:

```
./playSweeper.sh P5 REVTEST1
```

2 Design & Implementation

2.1 PEAS Model

Agent	Performance Measure	Environment	Actuator	Sensor
Logical Tornado Sweeper	Steps Taken, Win Rate	Configured 2D array world maps that simulate the Tornado Game	Probing, Making logical Inference, Marking tornadoes, Deciding next move	Numbers given on the coordinates, Positional coordinates

2.2 Problem Definition

This practical is about devising an agent that tries to use different strategies to win the game Tornado Sweeper, a variation of Mine Sweeper. Tornado Sweeper has similar rules and the only difference is that each cell has 6 neighbors. The board consists of N x N hexagonal cells, and the number on each cell indicates the number of danger in its neighbors (Figure 1 & Figure 2). On any map, two safe cells are given as the starting point: the top left and middle cell.

The task is to implement 4 probing strategies and an extended functionality (Reverse Tornado World Agent) for the agent to solve the puzzle by not only probing all the non-danger cells but also correctly marking all danger cells. The results will be compared and evaluated at the end of the report.

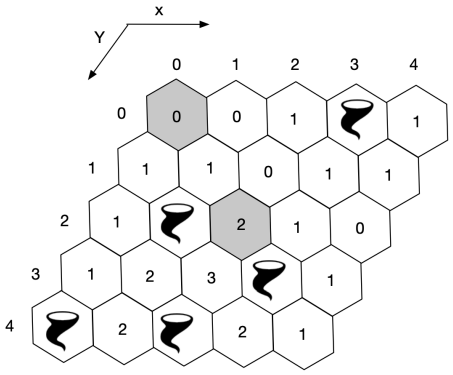

Figure 1. Visualization of the Tornado World

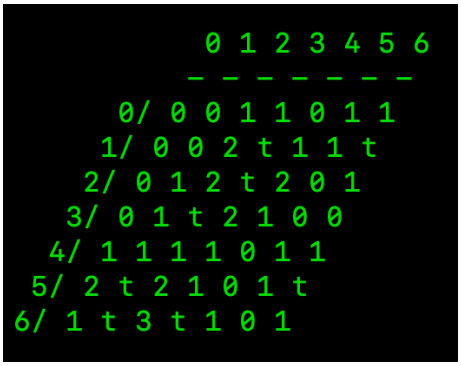

Figure 2. Implemented view of the Tornado World

2.3 System Architecture

All the files are in the same level and the same package, there are no sub-packages involved (Figure 3). Under /src, World.java stores all the maps as enum and A3main is responsible for getting request from the user and retrieving the appropriate map from World.java. It then starts a Game object that keeps the actual game map and an agent that has its own empty map to play on. A3main instructs Agent which strategy to use. Agent.java has all the probing-related methods and it interacts with Game to ask for information during probing and to check whether it has won the game. RevAgent is just a variation of Agent that plays the Reverse Tornado World and it also interacts with Game. The two agents tell A3main whether the strategy has worked, and A3main will produce the eventual output to the user.

The logical strategies used here require three libraries, and they're stored under libs directory.

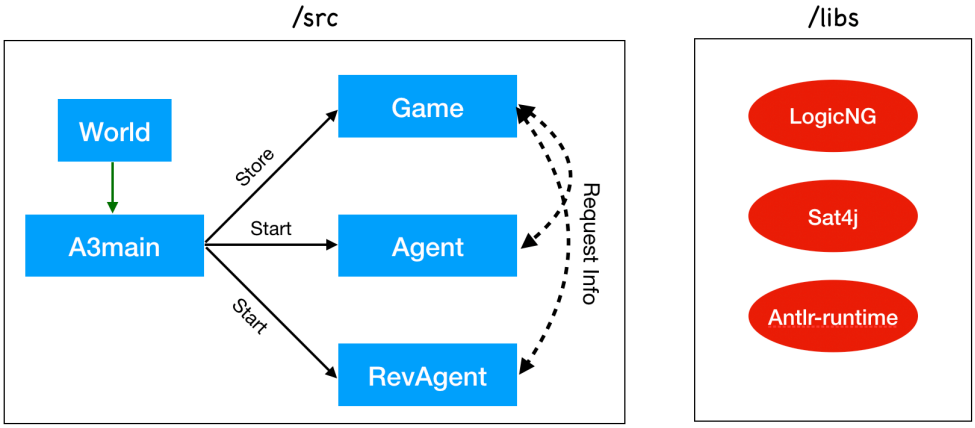

Figure 3. My System Architecture

2.4 Key Elements of Implementation

Four elements are worth mentioning in this part.

Utilizing Linked List:

In order to play the game where the map could constantly be altered, the agent could face a new view of map at each iteration and that means that cells that are previously undealable should be dealt with again to check for probing possibility. To realize this behavior, a Linked List is the perfect candidate. For all algorithms from Part 2 onwards, the agent initializes a 'frontier' linked list that stores all the coordinates. While the frontier is not empty, the algorithms pop one cell to check if it can be probed or marked. If nothing can be done with it, add it back to the queue so it can show up again after the map has been further discovered.

Detecting Stuck Situation:

In a situation where the game cannot be further proceeded, we might end up with an endless while loop because linked list can never be empty. To break the loop in this situation, an ArrayList of coordinates call 'trace' and a 'count' is initialized before the loop, it tracks all the undealable cells. At each iteration, if the cell is undealable, add it to 'trace' if not already in it. If it can be dealt with, it will be removed. This makes it possible to detect stuck situation because if the game is stuck, it will keep looping over the same undealable cells and no progress can be made. Whenever a cell is already in 'trace' and it's undealable again, we increment the count. If it's updated, reset the count to 0. This way if ever the count increments to equal the size of 'trace', this means that we have gone through all the undealable cells and no progress can be made the entire round, we're stuck. This then breaks the agent out of the loop (Figure 4).

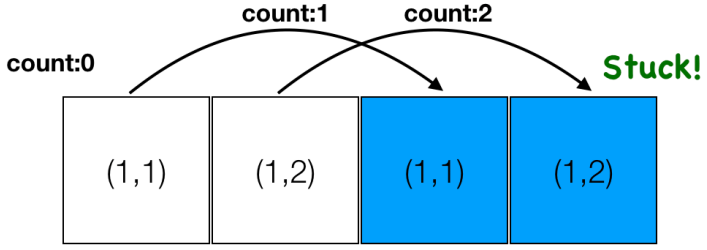

Figure 4. Stuck situation detector

Encoding Knowledge Base and Query:

For SAT algorithms, the agent constructs the logic sentence that represents knowledge base (`makeKB()`) of the current map view at each iteration, and adds "&~Dij" and "&Dij" to it to feed into the SAT solver as the query. The boolean results are sent back to the `probe()` stored in ifProbe or ifDanger to take action accordingly. The `makeKB()` first calls `makeClause()` for each uncovered non-zero cells to get the logical sentence from them each, then adds all the clauses up to one final sentence. There are two versions of `makeClause()`, one for CNF one for DNF. CNF clauses are constructed by combining the atMost() and atMostNon(), which is essentially atLeast(), of each cell based one their covered neighbors and remaining safe cell options. DNF clauses are encoded simply by constructing combinations of literals with appropriate operators.

Extension - Reverse Tornado World Agent:

For Part 5, the agent will adapt to the new game Reverse Tornado World, where the number on each cell represents how many non-danger cells are in the neighbor. This is particularly interesting because it's reverse thinking for human, where it might not be directly intuitive. However for AI, it's a simple change of code to adapt to the rule, the logical inference and even encoding can stay the same. The RevAgent uses DNF probing as default. In this version, the zero-cells are equivalent to the ones where their value equals the number of neighbors. The agent needs to learn this. We need an addition ArrayList to store the zero-equivalent cells, it's useful when constructing the knowledge base to exclude them. Lastly, the encoding part needs not to be changed, simply change the argument passed into the original makeClause() would do the work. The original second argument is (cell value - discovered dangers) which is danger in remaining neighbors. Now we make it (number of neighbors - cell value - discovered dangers), because (neighbors - cell value) is essentially number of

nearby dangers. This converts the logic back to the original version without actually changing the map.

3 Testing

My implementation achieved the following score during testing:

Folder Name	Total Tests	Passed Tests
/Tests	25	25
Test including P4&5	34	34

Example run showcase:

1. P3 on SMALL0

```
Agent P3 plays SMALL1


         0 1 2 3 4
        - - - - -
    0/ 0 1 t t t
   1/ 1 1 1 2 2
  2/ 2 t 1 0 0
 3/ t 2 2 1 0
4/ 1 1 1 t 1

Start!
Final map

         0 1 2 3 4
        - - - - -
    0/ 0 1 * * *
   1/ 1 1 1 2 2
  2/ 2 * 1 0 0
 3/ * 2 2 1 0
4/ 1 1 1 * 1
```

2. P4 on LARGE4

```
Agent P4 plays LARGE4


            0 1 2 3 4 5 6 7 8
           - - - - - - - - -
       0/ 1 2 t 1 1 1 1 t 1
      1/ 2 t 3 2 1 t 1 1 1
     2/ t 3 3 t 1 1 1 1 1
    3/ t 4 t 2 1 0 0 2 t
   4/ 1 2 t 2 0 0 0 1 t
  5/ 0 0 1 2 1 0 0 0 1
 6/ 1 1 0 1 t 1 1 1 0
7/ 1 t 1 0 2 2 1 t 1
8/ 0 1 1 0 1 t 1 1 1

Start!
Final map

            0 1 2 3 4 5 6 7 8
           - - - - - - - - -
       0/ 1 2 * 1 1 1 1 * 1
      1/ 2 * 3 2 1 * 1 1 1
     2/ * 3 3 * 1 1 1 1 1
    3/ * 4 * 2 1 0 0 2 *
   4/ 1 2 * 2 0 0 0 1 *
  5/ 0 0 1 2 1 0 0 0 1
 6/ 1 1 0 1 * 1 1 1 0
7/ 1 * 1 0 2 2 1 * 1
8/ 0 1 1 0 1 * 1 1 1
```

3. Detailed showcase of correctness (P3 on TEST2 verbose mode):

The following demonstration (Figure 5) shows the step-by-step logical inference process that the agent undergoes when playing on the 3x3 TEST2 Map. It first starts with probing the initial given safe cells (0,0) and (1,1). Then by translating the current world view into DNF encoded knowledge base, it decides whether the next cell in question is safe or dangerous to take the appropriate action. The process is 100% as expected and correct.

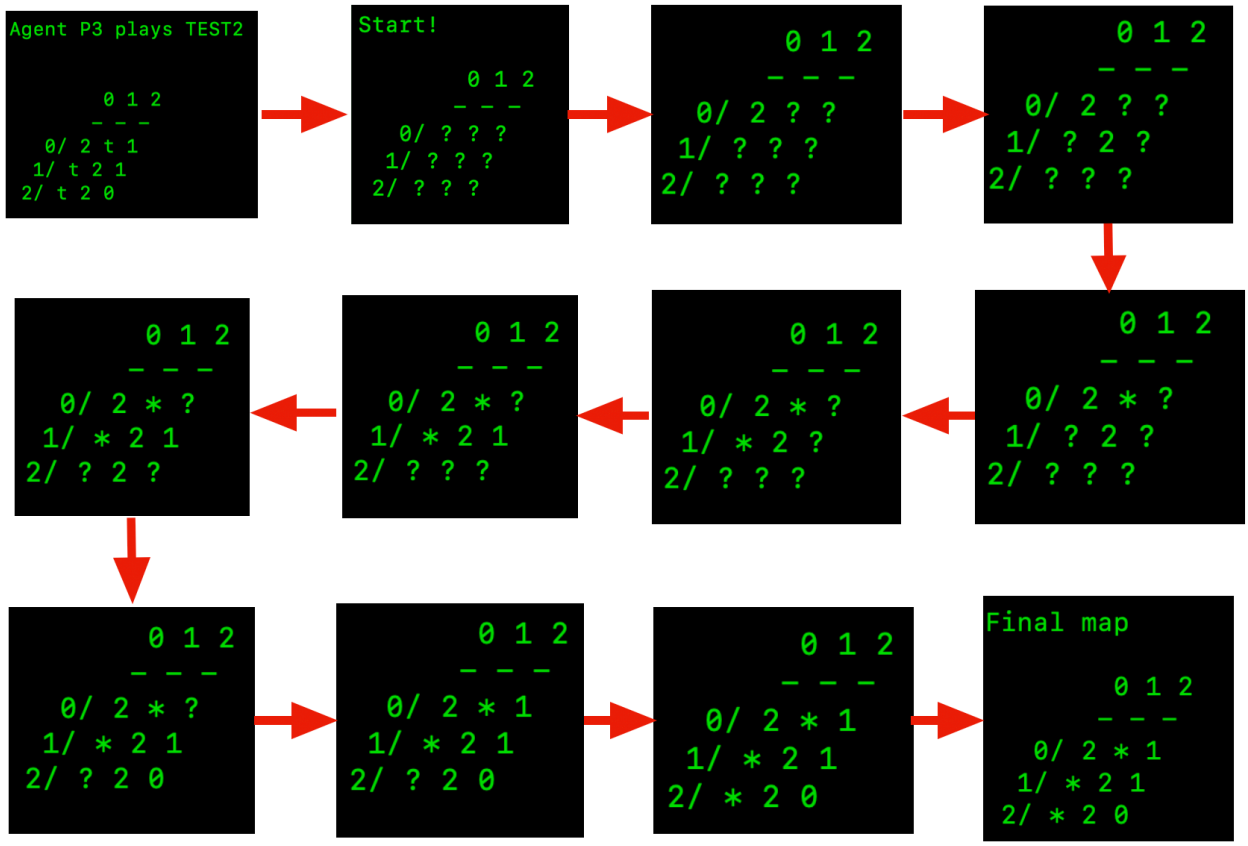

Figure 5. Demonstration of Step-by-step Agent process

4. P5 Reverse Tornado World showcase

To test the agent's adaptability to the new game rule, I have devised **three customized maps (REVTEST0,1,2)**, and below is its step-by-step process demonstration on one of the maps which works as expected (Figure 6).

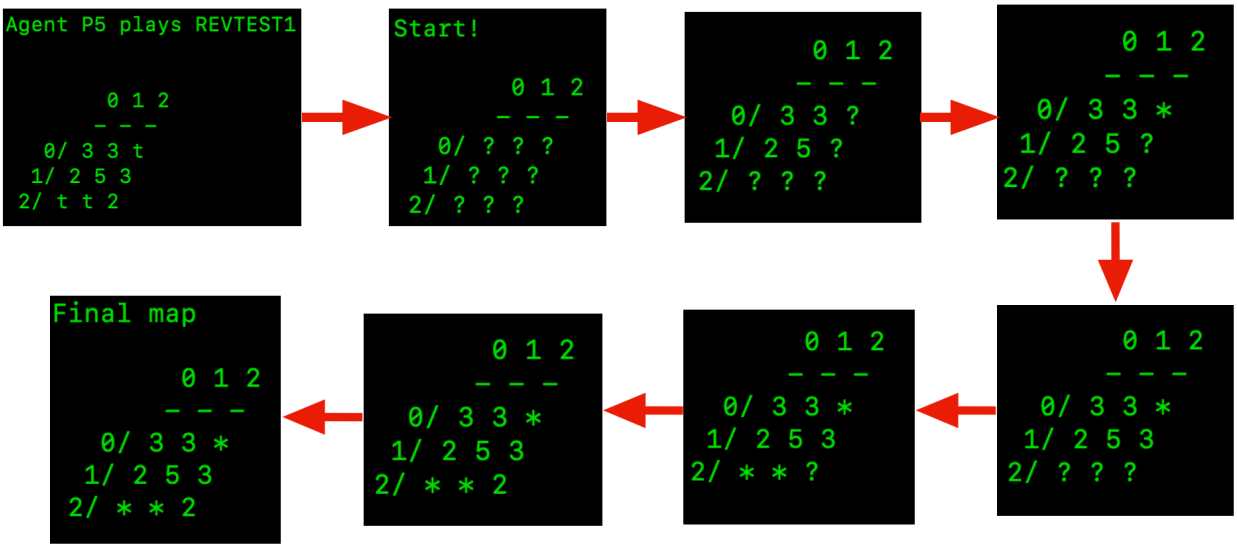

Figure 6. Demonstration of Step-by-step RevAgent process

4 Evaluations and Conclusion

4.1 Time & Space Complexity:

This implementation has places to improve in terms of time and memory cost. In each iteration, the agent constructs the knowledge base of the entire map just to do logical inference. This is O(n^2) time complexity. Most of the clauses in the knowledge base are irrelevant to make the inference on a cell in question, especially in bigger maps. For instance, checking SAT for (1,1), clauses for (20,23) might not be helpful at all. This leads to redundant memory occupied where information of 3 cells are required, but 100 were stored.

4.2 Agent Performance

Criterion 1. Accuracy - Win Rate

There are in total 36 maps (excluding the 3 for RevAgent), the three strategies have achieved the following scores in terms of winning the game.

Strategy	Solved Score	Small Maps	Medium Maps	Large Maps
SPS	10/36	3/10	2/10	1/10
SAT DNF	19/36	5/10	6/10	3/10
SAT CNF	19/36	5/10	6/10	3/10

The SAT strategy tackles more puzzles than SPS and is able to perform better in larger maps. However in general, all three strategies tend to decline in win rate as the map size grows.

Criterion 2. Guesses Taken

An efficient agent should be able to take as fewer guesses as possible to probe, mark and figure out the solution. Below is the guesses taken for each strategy on differently sized maps (for 5x5 maps, TEST0 is also included) (Figure 7), guesses taken is the average number calculated by total guesses divided by games won:

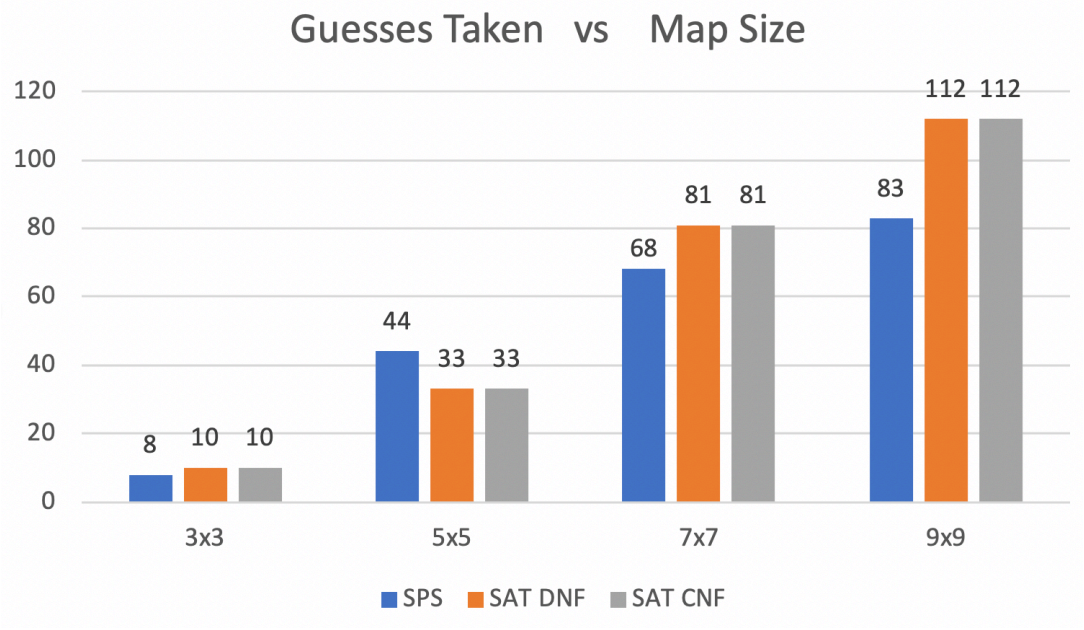

Figure 7. Average guesses taken by three strategies vs map sizes

As can be seen, the guesses needed by the algorithms to take to win a game increases as the size of the map increases. SAT strategies encoded by DNF and CNF have the same performance across the board, which is expected, because the only factor that would make a difference for guesses taken is the order of the cell being SAT tested. SAT strategies show stronger deductive power (fewer guesses) compared to SPS, the contradiction in 7x7 and 9x9 maps can be explained by the difficulty of the passed map itself. Since SPS barely passed any games on that level, the hard ones that naturally require more steps lead SATS to a higher average guess.

Conclusion:

In conclusion, the implementation of all 5 parts passed all the given test cases and have shown to work correctly as desired after step-by-step inspection. Using the SAT strategy, the agent is more competent to tackle difficult puzzles than SPS strategy and it takes fewer guesses. The agent is also capable of adapting to new rules and environment in the Reverse World to perform

effectively. In terms of places to improve, the time and space complexity could be reduced if at each iteration, the agent only constructs a partial knowledge base of the map that is necessary to make a logical inference for the cell at hand. This would save a lot of space and time. To improve efficiency in terms of guesses taken, different popping order of the cells to be dealt with can be experimented, this will reduce the occurrences where nothing can be done to a cell. For instance, the algorithm could start with the most promising areas.

5 Bibliography

Czengler (2022) “LogicNG - The Next Generation Logic Framework”. *LogicNG*. https://logicng.org/

Unknown Author (2021) “Strategy - MinesweeperWiki”. *MinesweeperWiki* https://www.minesweeper.info/wiki/Strategy

Sean Barrett (1999) “Minesweeper: Advanced Tactics”. nothings.org. http://nothings.org/games/minesweeper/

Baeldung (2020) “Java Bitwise Operators”. *Baeldung*. https://www.baeldung.com/java-bitwise-operators

Le Berre (2023) “Solver Documentation”. *Sat4j.org*. http://www.sat4j.org/maven233/apidocs/org/sat4j/minisat/core/Solver.html

Mun See Chang (2023) “The Danger Sweeper Knowledge Base”. https://studres.cs.st-andrews.ac.uk/CS5011/Lectures/L12-Logic3.pdf