Assignment 2

Saurav Jha

01 Nov, 2019

1 Program Overview

The submission intends to implement the basic and intermediate agents for the tornado sweeper game. The PEAS model for these agents can be defined as:

1. **Performance:** (a) The percentage of mines found (the greater, the better), and (b) The number of random moves (the lesser, the better).
2. **Environment:** The NxN grids of tornado worlds.
3. **Actuators:** Choosing a square to probe or flag.
4. **Sensors:** (a) the clue on the square choosen, and (b) uncovering a square with tornado and losing the game.

The program can be compiled and ran from *A2src/src/* as:

```
$ javac -cp .:../lib/junit-4.11.jar:../lib/junit.jar:../lib/net.
    ↪ sf.tweety.logics.pl-1.13-with-dependencies.jar:../lib/
    ↪ sat4j-pb.jar A2main.java
$ java -cp .:../lib/junit-4.11.jar:../lib/junit.jar:../lib/net.sf
    ↪ .tweety.logics.pl-1.13-with-dependencies.jar:../lib/sat4j-
    ↪ pb.jar A2main "SATX" "L9"
```

2 Design

The design for all three agents consider the nine points mentioned in the specifications. Each cell on the board is uniquely defined by its row and column numbers. The agents share a common knowledge about the world with techniques handful for navigating through it such as finding the neighbours, setting the visited and to-be-visited cells, the number of active tornadoes *(#T)*, etc. Further, they have their own copy of the world with all cells initially uncovered. All the agents start the game by probing cells at the top-left and the centre of the board before switching to their respective strategies.

2.1 Basic Tornado Sweeper Agent (RPX)

RPX employs the random probing strategy for exploring the tornado-sweeper board. A random row and a column values are generated until we get a combination that represents a cell yet to visited. On probing the cell, it processes the content of the cell, adds it to list of visited cells, removes from the to-be-visited list, and updates its own view of the board. If the cell contains a zero, the agent processes this to probe all the neighbours of this cell in a recursive fashion prior to selecting a further random move. Lastly, the agent prints reasonable information to let the viewer know of its current actions.

2.2 Intermediate Tornado Sweeper Agents

1. **SPX agent:** The SPX agent employs the single point reasoning strategy to explore the tornado sweeper world. The strategy can be summed up in the following points:

   (a) After first two steps, the agent starts exploring the board in a row-wise fashion.

   (b) On encountering a covered cell, it processes all its uncovered neighbours to extract their clues. It then scans the neighbours of the neighbour with clues to count: (a) the number of cells marked as danger (#D), and (b) the number of unflagged and covered cells (#C).

   (c) Based on the clue value of the neighbour and the extracted #D and #C values from neighbours of the neighbour, the agent checks for whether the cell has: (a) all marked neighbours (AMN), *i.e.* $\#C = clue - \#D$, and (b) all free neighbours (AFN), *i.e.* $clue = \#D$.

   (d) If any neighbour satisfies the AMN condition, the cell is flagged ('F') for a danger. In contrast, the cell is safely probed if it satisfies the AFN condition.

   (e) The loop breaks when the agent has successfully probed all but the number of tornado cells, i.e. #To-be visited cells > #T.

   Switch to RPX: If the board remains unchanged after an iteration, the agent switches to the random probing strategy to probe a covered cell in random, and return to single point strategy to explore the board again. If the randomly probed cell contains a tornado, it declares the game lost.

2. **SATX agent:** The SATX agent moves similar to SPX in that it follows row-wise fashion to explore covered cells on the board. However, it differs from SPX in the conditions for probing and flagging the cell. The agent's unique strategy is described below:

   (a) Similar to SPX, on encountering a covered cell, the agent processes its uncovered neighbours for the clues.

(b) For each uncovered and unflagged neighbour, a second-step neighbour-probing is performed to find all covered cells. Using the effective clue value (i.e. actual clue - # cells flagged) of each neighbour, a knowledge base (KB) of logic is prepared for each neighbour. The logic of all neighbours are then combined together to form a propositional formula (figure 1) to check: (a) KB $\wedge \mathrm{D}_{x,y}$ is unsatisfiable for probing, and (b) KB $\wedge \neg D_{x,y}$ is unsatisfiable for flagging the cell.

(c) Steps (a), (b), and (c) are repeated until all but T cells have been safely probed.

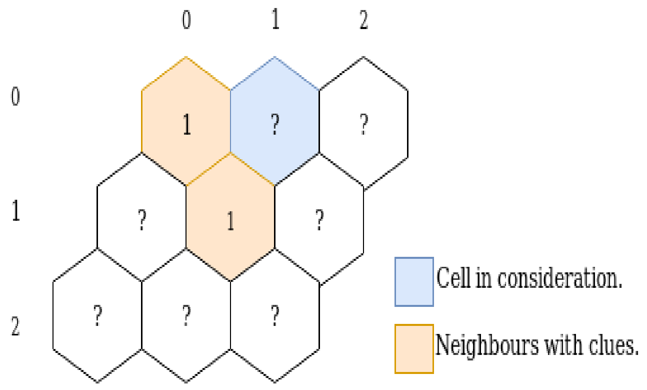

Figure 1: Figure showing the probing strategy for cell [1,1] of *TEST1* world. The combined propositional formula for both neighbours with clues is: ((!D0,1 D1,0) || (!D1,0 D0,1)) ((!D0,1 D1,0 D1,2 D2,1 D2,2) || (!D1,0 D0,1 D1,2 D2,1 D2,2) || (!D1,2 D0,1 D1,0 D2,1 D2,2) || (!D2,1 D0,1 D1,0 D1,2 D2,2) || (!D2,2 D0,1 D1,0 D1,2 D2,1)).

Switch to SPX: If the SATX agent fails to find a move satisfying the entailment, it switches to the SPX strategy to find the next move.

Switch to RPX: The switch to RPX agent is made when both SATX and SPX fail to find a move on the board.

3 Implementation

The program represents each cell on the board as an object of the class *Cell* with *row* and *column* attributes. The *getNeighbours()* method of cell returns all its valid neighbours in the board. All the agents inherit the *TornadoSweeper* class, which contains attributes for storing the original world and the agent's view of it, the *visited* and *toBeVisited* arrays, the number of tornadoes, the current step, and methods like *recursivelyProbeNeighbours()* to probe all neighbours of a cell with content *'0'* as long as all of of its neighbours are non-zeroes, *getFirstCellsToProbe()* to return the [0,0] or the centre cell to be probed in the first two steps of the game, *updateAgentWorld* to update the agent's world and add the cell to the *visited* array while removing it from the *toBeVisited* array. All the attributes of *TornadoSweeper* are set as protected to be shared among the subclasses of agents while being static so as to preserve the game's states while switching from one agent to another.

3.1 RPX agent

RPX agent uses the *generateRandomNumberInRange()* method to generate a cell not yet visited. If the cell selected is a tornado, it breaks the *gameLoop* to declare the game as lost. Otherwise, the *gameLoop* continues unless all but T cells have been visited.

3.2 SPX agent

The SPX agent moves through the board to find a covered cell and check the satisfiability for AMN and AFN conditions (section 1). *checkNeighboursofNeighbour()* returns count of the number of covered and flagged cells for each clue. *allMarkedNeighbours()* and *allFreeNeighbours()* utilize these counts to check for the condition to either flag or probe the cell, respectively. If flagged, the cell is added to the *tornadoCells* array. If the agent fails to find any move on the board, a random cell that has not been probed or flagged is picked to be visited. In case the cell is a tornado, the control is taken out of *gameLoop* to mark the game over before the termination condition, and the game is declared to be lost.

3.3 SATX agent

The SATX agent uses a combination of all its uncovered neighbours to construct the propositional formula. The formula is fed into the Tweety library to derive clauses in the conjunctive normal form (CNF). Individual literals and their negations are then extracted from the CNF and mapped to numerals to represent the DIMACS formula, e.g. if $D_{0,1}$ represented by 1 then -1 represents $\neg D_{0,1}$. The DIMACS form is fed to the SAT4J solver for satisfiability. Additionally, the isSatisfiabilty() method for SAT4J takes either the mapped positive or negative integer as argument for the cell:

1. For probing the cell, check if KB $\wedge \mathrm{D}_{x,y}$ is not satisfiable.
2. For flagging the cell, check if KB $\wedge \neg D_{x,y}$ is not satisfiable.

If both of above conditions are satisfiable, we leave the cell unchanged and move on to the next cell on the board.

4 Evaluation

The performance of each strategy for the world ”TEST1” has been reported for concise explanation. The world, as provided with the problem specification is:

	0	1	2
0	1	1	t
1	t	1	1
2	t	2	0

4.1 RPX approach

The RPX agent loses the game as soon as a cell with a tornado value ’t’ is probed.

1. **Case 1:** Game won after 2 random probes.

Table 1: Steps for RPX leading to a won game.

Step No.	Action
2	Randomly probe cell [0,1] to uncover clue ’1’
3	Randomly probe cell [2,2] to uncover clue ’0’. Probe its covered neighbours recursively: ([1,2]: ’1’), ([2,1]: ’2’)
4	All but 3 cells are safely uncovered. Declare game won!

2. **Case 2:** Game lost in just 1 random probe.

Table 2: Steps for RPX leading to a lost game.

Step No.	Action
2	Randomly probe cell [0,2] to uncover clue ’t’. Declare game lost.

4.2 SPX approach

1. **Case 1:** Game won. For winning a game, minimum number of random probes required for *TEST1* is 1, which applies to the example in table 2, while the maximum number of random probes is 2.

2. **Case 2:** Game lost after 2 random probes.

4.3 SATX

1. **Case 1:** Game won in 1 random probe.

2. **Case 2:** Game lost after 1 random probe.

Table 3: Steps for SPX leading to a won game.

Step No.	Action
2	Explores the entire board to pick a cell with all-free or or all-marked neighbours. Finds none.
3	Switches to RPX to randomly pick and probe cell [2,2] with content ’0’. Probe its covered neighbours recursively: ([1,2]: ’1’), ([2,1]: ’2’). Switches back to SPX.
4	Finds [1,0] satisfying AMN: flags for danger.
5	Finds [2,0] with AMN: flags for danger.
6	Finds [0,1] with AFN. Uncovers for content ’1’.
7	All but 3 danger cells have safely been explored. Declares game won.

Table 4: Steps for SPX leading to a lost game.

Step No.	Action
2	Explores the entire board to pick a cell satisfying AFN or AMN. Finds none.
3	Switches to RPX to randomly probe [1,2] with content ’1’.
4	Finds no cell satisfying AFN or AMN on the board.
5	Switches to RPX to probe [1,0] with content ’t’. Declares game lost.

Table 5: Steps for SATX leading to a won game.

Step No.	Action
2	Visits board to check for entailment. Finds none.
3	Switches to SPX and then to RPX to finally probe [2,2] with ’0’. Probes neighbours ([1,2]: ’1’), ([2,1]: ’2’)
4	Finds KB $\wedge \neg D_{1,0}$ unsatisfiable. Flags cell.
5	Finds KB $\wedge \neg D_{2,0}$ unsatisfiable. Flags cell.
5	Switches to SPX. Finds [0,1] satisfying AFN. All but 3 cells probed, declares game won.

Table 6: Steps for SATX leading to a lost game.

Step No.	Action
2	Visits board to check for entailment. Finds none.
3	Swi
12	Finds KB $\wedge \neg D_{1,0}$ unsatisfiable. Flags $D_{1,0}$.
20	Randomly probes [2,0] with ’t’ after moving through the board.

4.4 No. of Random Probes

On average, the SATX agent requires far lesser random probing in average than that of SPX (see Table 4.4). This gives SATX an upper hand in winning the game. The number of random probes is further determined by the number of cells with a '0' clue. Thus, all the agents have greater chances of winning in the boards with more '0' cells as these significantly reduce the search space for random probing.

Table 7: Average no. of random probes for winning a game.

World	SPX	SATX
S10	3	1
M10	1	0
L1	0	0
L5	3	2
L9	1	1

5 Testing

Apart from evaluation on the given worlds, the program implements unit test cases included within the *src/* directory.[1] The file *CellTest* tests desired behaviour of each cell, *TestTornadoSweeper* tests these for the parent class while *AgentTests* tests the core behaviour of all three agents. These can be run by including the same class paths as in case of main file.

[1]Test files have in included in *src/* because of dependency issues for including in *tests/* while running on the lab machines.