CS5011 - Practical 2

[190026921]

Mar 09, 2022

Word Count: 2150

Contents

1 Introduction and Overview

The objective of this practical was the implementation of logical agents to play the Minesweeper adaptation, Obscured Sweeper. By making logical deductions, the agents clear a board by probing cells that do not contain mines. Some cells further are obscured, i.e., do not contain a hint (but also no mine).

1.1 Implementation status

All requirements were implemented, including the advanced agent.

- **P1:** Attempted & fully working (A/FW)
- **P2:** A/FW
- **P3:** A/FW
- **P4:** A/FW
- **P5:** A/FW; agent can play on triangular and hexagonal grids

1.2 Compiling and running instructions

1.2.1 Basic usage

As per the specification, the program can be compiled and run with provided `playSweeper.sh` script from the `src` directory:

```
./playSweeper.sh <Agent> <World> [verbose] [<world-mode>]

# or if already compiled
java A2main <Agent> <World> [verbose] [<world-mode>]
```

where

```
Parameter    Meaning            Options
=========================================================================
<Agent>      Agent              P1 | P2 | P3 | P4
<World>      World              TEST1 | ... | SMALL | .. | MEDIUM | ..
<verbose>    Verbose output     verbose
<world-mode> Mode for Advanced  rect | tri | hex
```

To use the advanced agent (P5), specify `tri` or `hex` as world-mode with any of the agents. See the `TriWorld` and `HexWorld` classes to see which worlds are available.

1.2.2 Instructions for testing and evaluation

To run the JUnit tests and the evaluation, compile the project with the Make commands from the provided Makefile.

All Make commands must be run from the project root (containing the `src` directory) and `make compile` must be run first to compile the project.

The following commands are available:

`compile`	`Compile all java files`
`clean`	`Remove java class files`
`test`	`Run JUnit tests`
`stacscheck`	`Run stacscheck checks (only on school server)`
`evaluation`	`Run the evaluation (run experiments, create plots and tables)`
`help`	`Print available commands`

Note that if you only want to run the game without the tests, you may run it as described above without the Makefile. The graphs are generated with a python script which uses the `pandas` and `matplotlib` libraries specified in `evaluation/requirements.txt`).

2 Design and Implementation

2.1 PEAS model and problem definition

The PEAS model can be applied to the problem at hand as follows:

- **Performance measures:** Two performance measures are used for the agent. The number of iterations measures how quickly an agent reaches its goal. The percentage of uncovered cells remaining at the end of the game measures the effectiveness of the agent.
- **Environment:** The game is played on an $N \times N$ board where cells either contain a mine, a clue denoting the number of adjacent mines, or are blocked (do not contain mine or clue). Uncovering a cell with a mine kills the agent and ends the game. Uncovering a cell with clue 0 automatically uncovers all adjacent cells. The top-left and center cells never contain a mine. The agent cannot see the content of a cell until it has been probed (except for blocked cells, which are always visible).
- **Actuators:** The agent can uncover cells by probing them to reveal their content.
- **Sensors:** The agent can sense the number of mines adjacent to any uncovered, non-blocked, non-mine cell through the clue provided on the cell.

With this, the problem can be stated as follows: Given a world, the agent must probe the cells of the world to find the location of all mines without probing any of them.

2.2 System architecture

The project is split into three top-level packages. The `delegate` package contains the parent class of the game, `ObscuredSweeper`. The package `logic` contains all classes responsible for the logical strategies of the agents. The `model` package contains classes related to the objects holding information about the game, such as the agent and the world.

All agents inherit from an abstract `Agent` class. This yields the `BasicAgent` (P1), the `SpsAgent` (aka Beginner Agent, P2), as well as the `SpsAgent`. The latter can either make use of a `DnfKnowledgeBase` (Intermediate Agent, P3) or a `CnfKnowledgeBase` (Intermediate Agent, P4). By default, games are started with a rectangular world (`RectWorld`) which is then passed as a view to the agent. However, all agents may also act on a view of `TriWorld` or `HexWorld` (Advanced Agent, P5).

Note that I renamed the `World` enum to `RectWorld` and created a separate interface `World` which is implemented by the different world types (rect, tri, and hex).

Figure 1 contains the class diagram showing this in more detail.

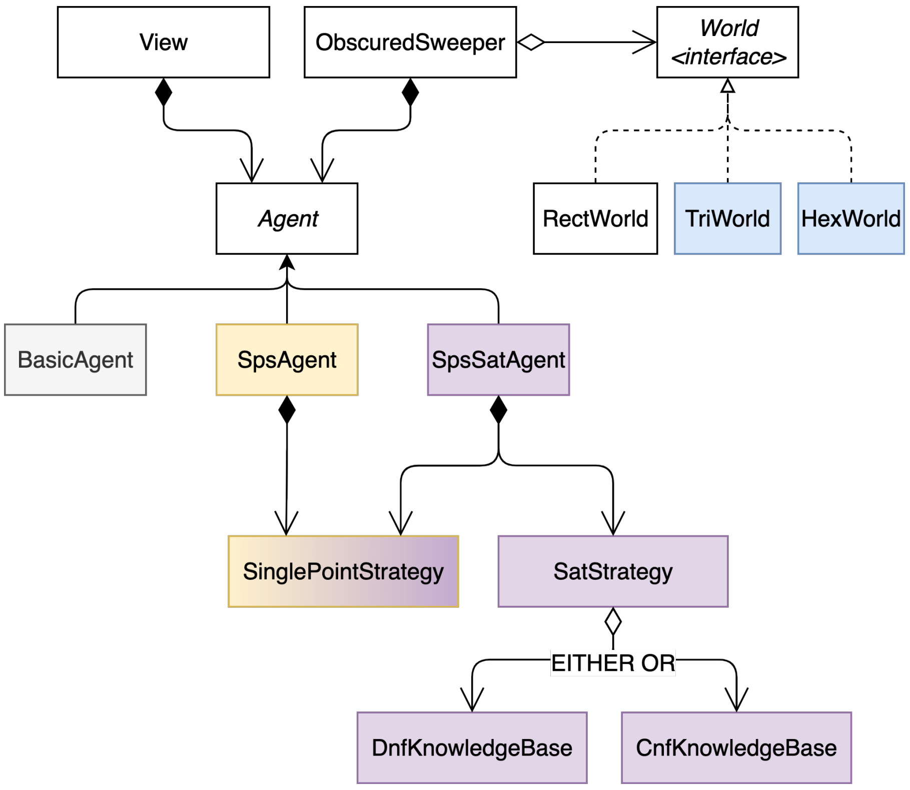


Figure 1: Class diagram showing the main classes used in the project. Classes related to the Basic Agent are colored in gray; classes of the Beginner Agent are colored in yellow; classes of the Intermediate Agents (DNF and CNF respectively) are colored in purple; classes of the Advanced Agent are colored in blue.

To implement the different encoding, the `SpsSatAgent` has either a `DnfKnowledgeBase` or a `CnfKnowledgeBase` associated with it. These knowledge base classes create a `DnfEncoder` instance, or `CnfEncoder` instance, respectively. Each knowledge base also takes a `View` object. I chose this differentiation between `View` and `KnowledgeBase`, since all agents require a view of the game world, but not all (e.g., `BasicAgent`) need access to the encoding strategies provided through the `KnowledgeBase`.

Figure 2 shows the class diagram for the encoders and knowledge bases.

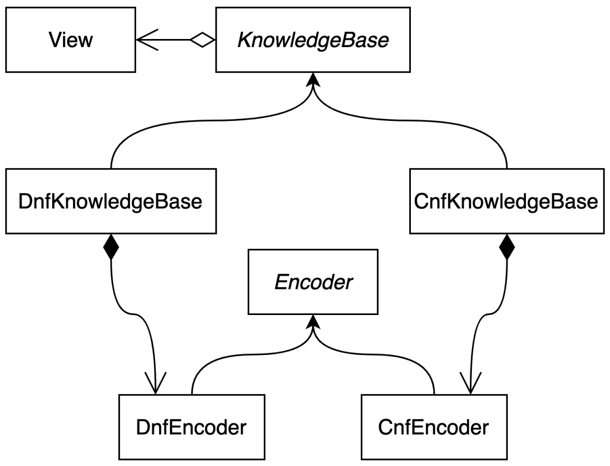


Figure 2: Class diagram showing the `Encoder` and `KnowledgeBase` classes.

2.3 Key implementation details

2.3.1 Game flow

A game is started on an `ObscuredSweeper` instance by calling the `run` method. From there, the agent will be asked to play (`Agent.playGame`) until one of three things occurs.

1. The agent probes a mine. In this case, a `MineFoundException` is thrown to exit the agent's `playGame` method early. This will only happen for the `BasicAgent`, since the other agents never probe cells with a mine.
2. The agent has no more cells to probe with its strategy, yet there are still uncovered cells left. A `NothingToProbeException` is thrown.
3. The agent uncovers all cells without probing a mine. No exception is thrown.

The `playGame` method of the `SpsSatAgent` is the most evolved implementation, combining the single point and SAT strategies (see Algorithm 1). The method runs a while loop as long as there are unknown cells left to probe and there was a change (i.e., flagging or probing) in the previous iteration. The second condition ensures that the algorithm terminates if the agent can make no more deductions.

In each iteration of the while loop, the method iterates over a collection of remaining unknown cells. For each cell, the agent employs the `check` methods of the single point and SAT strategies (see Algorithm 2). These methods determine if the current cell is either safe to probe or contains a mine, in which case the cell is flagged. Since the strategies should never disagree on a certain deduction about the content of the cell, it suffices to stop as soon as either of the two makes a change.

After the while loop terminates, the algorithm evaluates if the agent terminated. If not, a `NothingToProbeException` is thrown to signal that the agent did not terminate.

Algorithm 1 Procedure of the `SpsSatAgent.playGame` method

```
procedure SpsSatAgent.playGame
    initialSafeProbes()
    changed = true
    while unknownCellsLeft() & changed do
        changed = false
        for all cell in getUnknownCells() do                ▷ Check SPS then SAT
            if SinglePointStrategy.check(cell) then changed = true
            else if SatStrategy.check(cell) then changed = true end if
        end for
    end while
    if !changed & unknownCellsLeft() then                   ▷ Agent can't terminate
        throw NothingToProbeException
    end if
end procedure
```

2.3.2 Single Point Strategy

The `SinglePointStrategy.check` method (see Algorithm 2) performs the all-free-neighbors (AFN) and the all-marked-neighbors (AMN) checks.

The AFN check for a given cell `x` returns true, if any neighbor of `x` that contains a clue already has `x.clue` dangers uncovered around it. In this case, it is safe to probe `x`.

The AMN check returns true, if any neighbor of `x` that contains a clue has exactly `x.clue` + `x.dangerNeighborCount` unknown cells in its neighbors. In this case, `x` is flagged.

2.3.3 SAT Strategy

Similarly, the `SatStrategy.check` method checks if the knowledge base entails the cell being a danger, and if the knowledge base entails the cell NOT being a danger. These entailment checks can be performed either with DNF encoding or with CNF encoding.

Algorithm 2 `SinglePointStrategy.check` and `SatStrategy.check` methods

```
procedure SinglePointStrategy.check(cell)
    if allFreeNeighbors(cell) then
        probe(cell) return true
    end if
    if allMarkedNeighbors(cell) then
        flag(cell) return true
    end if
end procedure

procedure SatStrategy.check(cell)
    if entails(cell) then                  ▷ check if $KB \models$ Danger
        probe(cell) return true
    end if
    if entails(!cell) then                 ▷ check if $KB \models$ not Danger
        flag(cell) return true
    end if
end procedure
```

DNF encoder

The knowledge base in the DNF case is built in the `DnfKnowledgeBase.getKBU` method.

The method iterates over all uncovered cells with a clue and at least one uncovered neighbor. For each of these cells the `DnfEncoder.encode` method is called with the cell and the number of still covered adjacent dangers of the cell (call it k). For example, if cell `x` has clue 3 and one mine in its neighbors has already been flagged, there must be $k = 2$ more remaining dangers in the unknown neighbors of `x`.

`DnfEncoder.encode` returns the DNF of the cell as a `LogicNG.Formula`. The conjunction of the result of all cells is returned as the KBU.

What does the `DnfEncoder.encode` actually do? The result of the `encode` method is a disjunction of clauses representing all combinations of the neighbors of `x` such that exactly k cells in each combination contain a danger.

The combinations of cells are created using the `Encoder.booleanPermutations` method. This method generates all permutations of n boolean values, such that exactly k of them are true. I implemented this using Heap's algorithm. The list of cells passed to the `encode` method are then mapped to these boolean values.

The entailment check occurs in `DnfKnowledgeBase.checkEntailment`. The string representation of the cell to be checked is converted to a `LogicNG.Literal` and fed to a SAT solver together with the current KBU. If the negative of a cell (i.e., entailment of `not DANGER`) is checked, this is represented in the literal.

CNF encoder

The CNF based knowledge base follows a similar procedure.

When building the KBU, the `CnfEncoder.exactly` method is called with the cell `x` and the number of remaining dangerous neighbors k (see last section). This is to encode that exactly k more neighbors of `x` are dangerous.

The `CnfEncoder.exactly` wraps calls to the `CnfEncoder.atMost` and `CnfEncoder.atLeast` methods. Both methods in turn call `CnfEncoder.encode`, which uses a binomial encoding strategy (similar to that described by Frisch and Giannaros [2010]), to compute the CNF encoded constraint.

To encode the `atMost` k of n variables constraint, this entails considering all $\binom{n}{k+1}$ binomial combinations of the n variables. For the `atLeast` k of n variables, there are $\binom{n}{n-k+1}$ such combinations.

The `CnfEncoder.generateBinomialCombinations` method generates these binomial combinations for a generic list of integers from 0 to $n-1$. The solver requires DIMACS format, so to map the cells to these combinations, each cell is converted into its unique integer identifier (`identifier = row * width + column + 1`). In the generic binomial combinations, the 0 then maps to the integer identifier of the first cell in the variables to encode, 1 maps to the second one, and so on. In the `atMost` case, the integer identifiers are negated during the mapping as per the DIMACS format.

Lastly, to check entailment, the `DnfKnowledgeBase.checkEntailment` method adds all subclauses of the KBU, together with the DIMACS CNF encoded cell, to the `Sat4J` solver. The method returns true if the problem is satisfiable and no contradiction occurs, i.e., the KBU entails the proposition.

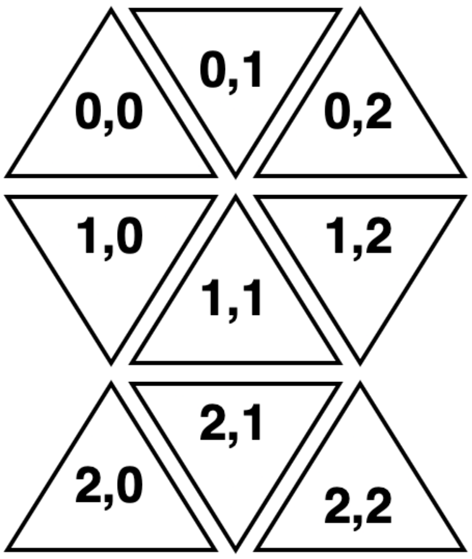
2.3.4 Advanced agent

For Part 5, I implemented the agents' ability to explore worlds on triangular or hexagonal grids. The coordinate system for both grid systems is provided in the sub-figures of Figure 3.

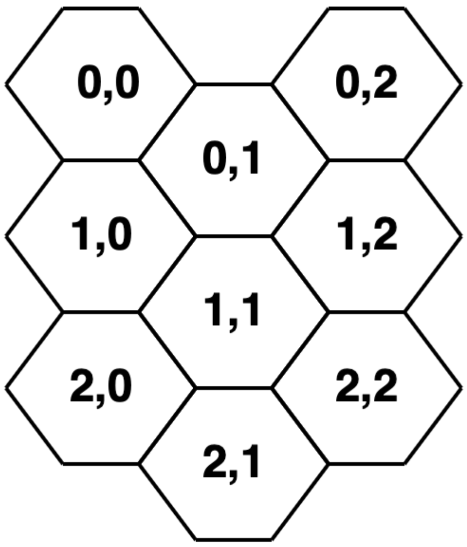
To allow an agent to explore tri and hex-grids, a specific method for each grid-type to get the neighbors of an adjacent cell was required. This is the only grid layout dependent function that the agents access.

I further implemented the `TriWorld` and `HexWorld` enums. They correspond to the `RectWorld` enum of the starter code, but provide worlds for the respective grid type. The methods for this were implemented in the `TriGBU` and `HexGBU` classes that extend `GameBoardUtils`, which is used to perform calculations on the character maps of the worlds and views.

For each, three worlds were implemented (see Figure 6, Appendix).



(a) Triangular grid.



(b) Hexagonal grid.

Figure 3: Triangular and hexagonal grids of the advanced agent.

3 Test Summary

I tested my application with extensive unit tests written in JUnit. Overall, I wrote 70 unit tests with a total line coverage of 97.6%. The coverage report is presented in Table 4 in the Appendix and in the `doc/coverage` directory of the submission. To detect breaking changes early, I set up a continuous integration workflow using GitHub Actions, where my repository is hosted (privately). The workflow is defined in `.github/workflow/main.yml`.

Beyond this, I used `stacscheck` with the 19 provided test cases.

4 Evaluation and Conclusions

4.1 Performance metrics

As discussed in the PEAS model, the agents can be evaluated in terms of number of iterations as well as percentage of unknown cells remaining. Figure 4 depicts this metric for the rectangular worlds for agents P1 to P4. Table 1 reports the average and standard deviations of the metric.

The data from the table clearly demonstrates that the more advanced agents are more effective at solving the game. The difference is especially stark when comparing P3 and P4 to P1, however, even the single point strategy (P2) can solve more than 60% of cells on average.

In terms of the number of iterations, comparing the metrics across the board is somewhat less useful, since an agent may achieve a low number of iterations only due to dying (or getting stuck) early. This is especially noticeable for P1 who probes a mine in all evaluated cases, thus

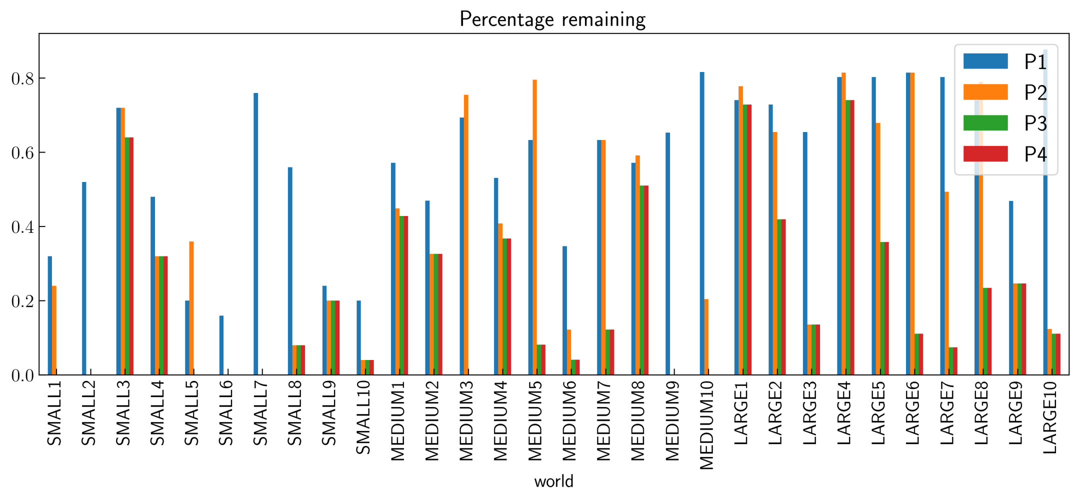


Figure 4: Percentage of unknown cells remaining after agent has finished on the regular, rectangular worlds.

Table 1: Mean and standard deviation of percentage remaining on rectangular worlds

agent	**mean**	**std**
P1	0.58	0.21
P2	0.39	0.29
P3	0.21	0.23
P4	0.21	0.23

dying early. Figure 5 depicts the metric for the rectangular worlds, and Table 2 reports the averages and standard deviations.

Table 2: Mean and standard deviation of number of iterations for rectangular worlds

agent	**mean**	**std**
P1	2.4	1.38
P2	2.57	1.59
P3	2.6	1.16
P4	2.6	1.16

Unsurprisingly, P1 uses the least iterations (on average), since it dies early on. P2 achieves a slightly lower average number of iterations than P3 and P4. Recall that P3 and P4 manage to explore more of the board (lower percentage remaining), which now manifests in a higher number of iterations. The increased effectiveness thus comes at the price of more iterations and a longer run-time.

For triangular and hexagonal worlds, the same analysis could be performed (and the plots and tables are in fact created when the `make evaluation` command is run). However, since only

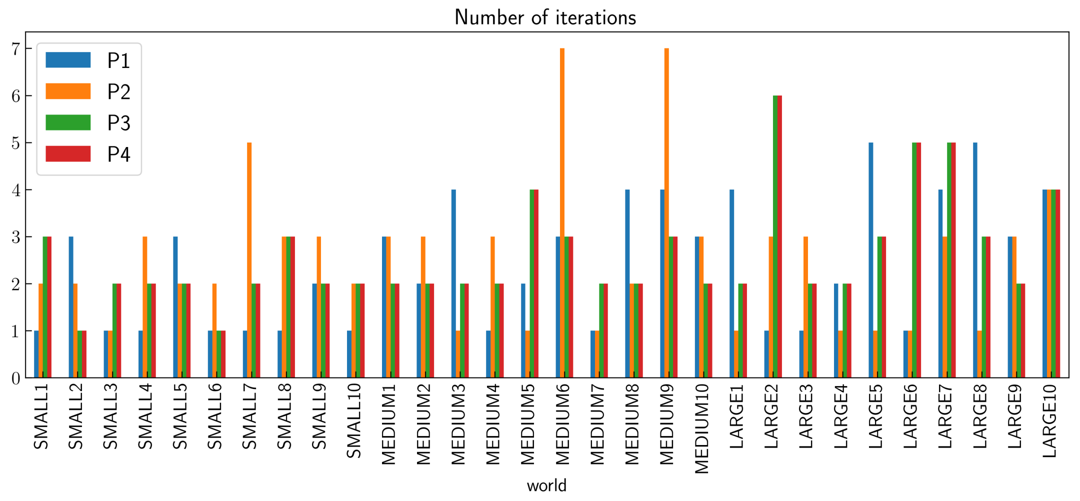


Figure 5: Number of iterations, rectangular worlds.

three worlds were evaluated per grid type, the results should be taken with a grain of salt and are not generalizable, hence they are not presented here.

4.2 A note on CNF vs DNF encoding

As was apparent in the previous subsection, P3 and P4 produce the same results. This is no surprise, as the two agents' different encoding strategies do not influence the logical deductions.

From an implementation point of view, the DNF encoding proved to be easier since the `LogicNG` library takes care of much of the difficult leg work. Since we were restricted on which methods to use in the `Sat4J` library, CNF was more difficult. However, if one were to make use of the entire functionality provided by standard libraries such as `Sat4J`, I assume there would be little difference in terms of complexity of implementation.

Table 3: Mean and standard deviation of run-time / iteration (in ms) on rectangular worlds. Experiments were run on a MacBook Pro (2019) running a 1.4 GHz Quad-Core Intel Core i5.

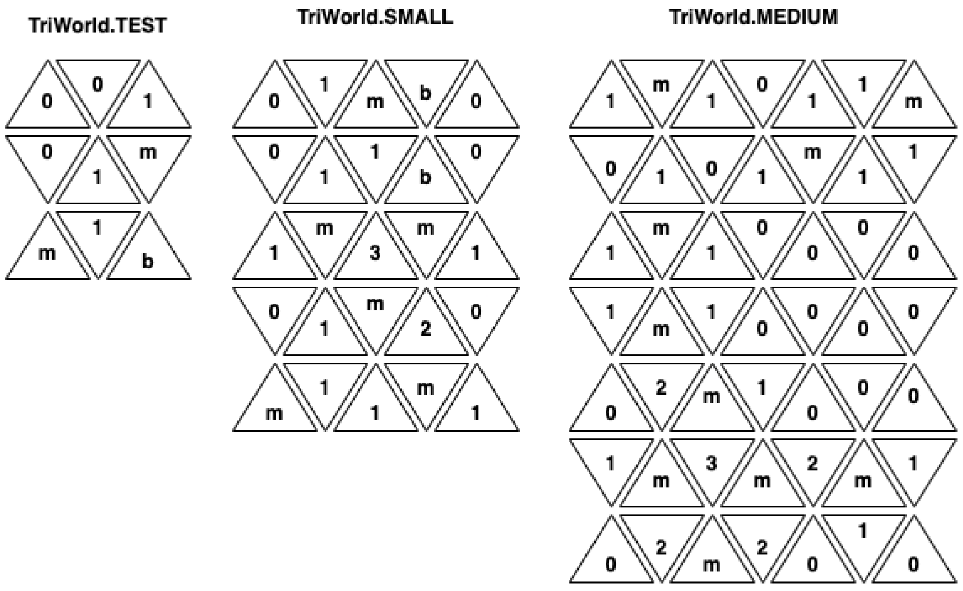
agent	**mean**	**std**
P1	0.15	0.53
P2	0.2	0.32
P3	17.62	53.04
P4	4.18	4.84

That being said, consider Table 3, which depicts the time taken per iteration in milliseconds. Here, P4 is approximately four times faster than P3. This could suggest that the direct use of CNF encoding is faster than DNF. However, this is also highly dependent on the implementation details, which could account for the difference in run-time. A more detailed evaluation of this is out of the scope of this report.

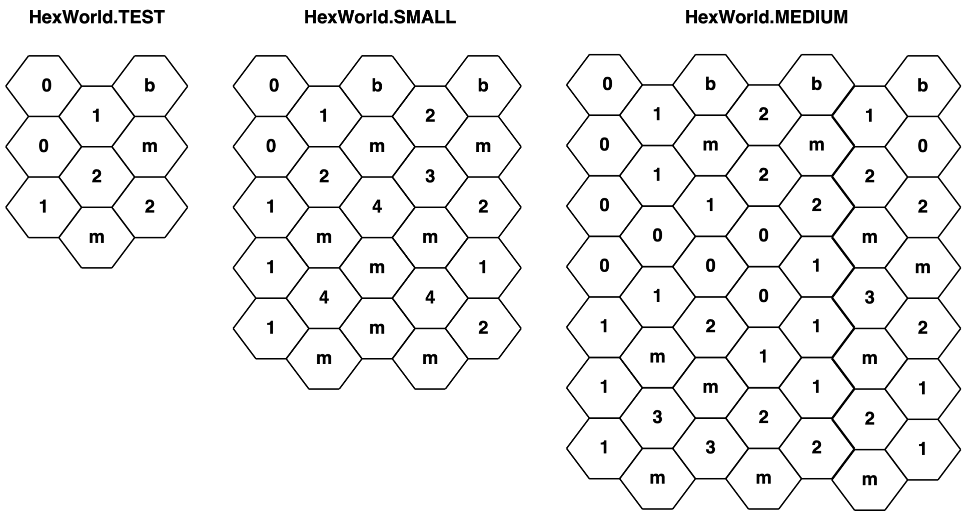
5 Appendix

Table 4: Test coverage report of the JUnit tests for the application. Overall, 97.6% of classes (40/41) were tested, 93.6% of methods (250/267), and 96.3% of lines (1064/1105).

Package	**Class**	**Method**	**Line**
delegate	100% (2/2)	97% (32/33)	99.2% (131/132)
logic	100% (14/14)	100% (81/81)	99% (378/382)
model	100% (1/1)	55.6% (5/9)	69.2% (9/13)
model.agent	100% (9/9)	98.1% (51/52)	99.1% (233/235)
model.agent.exceptions	100% (2/2)	100% (2/2)	100% (2/2)
model.board	92.3% (12/13)	87.8% (79/90)	91.2% (311/341)



(a) Triangular worlds.



(b) Hexagonal worlds.

Figure 6: Triangular and hexagonal worlds of the `TriWorld` and `HexWorld`.

References

A. M. Frisch and P. A. Giannaros. Sat encodings of the at-most-k constraint: Some old, some new, some fast, some slow. 2010.