University of St Andrews

School of Computer Science

CS5011 Artificial Intelligence Practice Practical 2

Parts attempted: 1 and 2

ID: 170018405

Word Count: 4866

Table of Contents

Table of Figures

1.0 Introduction

Minesweeper is a 1960s single-player computer game designed by R.Donner, and C.Johnson, which consists of an n by m grid of cells containing mines and clues to the spatial locations of the former distributed throughout a board-style map. 1991 marks the year when the game gained popularity, when Microsoft decided to include it as part of its Windows operating system (Castillo & Wrobel, 2003). The basic task of the game is to challenge a player with uncovering all the cells which do not contain mines while avoiding those that do. While the challenge appears deceivingly simple, it actually requires players to employ a combination of logical, arithmetic, and probabilistic reasoning techniques to solve a given world. It is in fact this very reason that this game caught the attention of both AI researchers and mathematicians as they attempted to tackle the challenge of developing algorithms that could solve a given world as efficiently as possible. While there was some debate initially, it was quickly identified that Minesweeper was indeed an NP-complete problem (Kaye, 2000). However, as many have pointed out, this does not mean that the game is easy to solve (Scott, Stege, & Rooij, 2011). Most of the solutions proposed for this game thus utilise some form of AI agent. From the published works, the majority of solutions employ a form of single point strategy (SPS) technique combined with another strategy – typically involving logic (David, 2015). The majority of the solutions thus stem from the Symbolic branch of AI, utilising logical propositions to come up with solutions. Kaye for example showed how CNF formulas written for minesweeper expressed as logic circuits can be used to tackle problems that cannot be satisfied with the single point strategy. On the other hand, other academics have used constraint satisfaction approaches (Bayer & Snyder, 2006), linear algebra (Massaioli , 2013), and even learning techniques (Castillo & Wrobel, 2003) to solve minesweeper worlds. The game has thus undoubtedly served as a test bed for improving AI algorithms much like the Wumpus world described by Russell and Norvig (Russell & Norvig, 2016).

The purpose of this report is to build agents to solve a game that is a subset of Minesweeper known as Daggers and Gold (D&G).

2.0 Problem decomposition (PEAS model)

The first step in the design of an agent system is usually commenced with a thorough description of the environment which an agent must act upon. Through this description, the tools that can be made available to the agent to achieve its goals can be well defined. One way to proceed with such a description is to utilise the PEAS (Performance, Environment, Actuators, and Sensors) model as described by Russell and Norvig (Russell & Norvig, 2016).

The environment is the starting point for the problem decomposition. In the case of the D&G world, the environment consists of a square grid of covered cells each containing a string depicting its contents. The cells' contents consist of either daggers, gold mines, or a number which indicates the sum of cells containing daggers adjacent to the cell being examined. The environment also has certain behaviours (or laws) that dictate how it behaves as an agent acts upon it. These behaviours include:

- The environment displaying a cell's contents once an agent probes (uncovers) the cell in question or a cell that automatically reveals this cell's contents.
- Cells in the environment that contain either a value of 0 or g (for gold mine) reveal all adjacent cells if probed by an agent.
- In the case where a gold mine is probed by an agent, the agent receives an extra life, whereas if an agent probes a dagger, the agent loses a life.
- The environment therefore keeps track of the total life count of the agent, and thus has logic to determine the game state (won, lost, or in-play) at any step during run-time.
- The game won (level solved) state is returned only once all cells except those with daggers in them have been uncovered.

The environment of the D&G world according to the above description can therefore be described as discrete, static, and single agent (as the internal states of the environment i.e. the contents of the cells never change, and, only one agent at a time is capable of performing actions at any step. The environment may also be described as sequential given that rewards such as gaining a life, or winning the game only come after a series of actions taken by the agent. Moreover, a partially observable label may be attributed since an agent can only perceive information about the cells is has uncovered so far. However, whether the environment is known or not, is a point of debate. It may be argued that give the immutability of the D&G world, then the transition model for the environment is completely known. This means that if an agent can deduce that a cell has no dagger within it, then it knows that the outcome of probing this cell will not lead to a reduction in life count, and that the cell in question will be uncovered. However, what should be realised here is that while the agent may know if a cell contains a danger, it does not necessarily know how many cells will be uncovered once a cell is probed. Additionally, the agent may also reach a point where it cannot deduce what a cell contains through its perception of the world, and has to resort to random probing. This leads to the conclusion that the D&G world is semi-known.

The second phase of the problem decomposition is achieved by providing a fitting description of the performance expected of the agent. To start with the most obvious score for an agent to track its performance is the life count it has acquired through the discovery of gold mines (+1 life) and probing of daggers (-1 life). This score is important to track as it has a direct impact on whether an agent wins or loses the game. Yet, this score alone is not sufficient to allow an agent to play a D&G game in a desirable manner. This is because the agent as of yet has no concept of a goal state. Thus, the next important reward to peruse, is that of winning the game by uncovering all the cells that do not contain daggers within them. It could hence make sense to reward the agent for probing a cell. However, this is not desirable as the agent could just continue to probe safe cells it has previously probed to gain a higher score. Hence, in reality, the score of the agent must be decremented as a form of cost every time the agent probes a cell. This would intern mean that the agent is forced to make the least number of probes to win the game, which is desirable in terms of efficiency. This combined with the reward of gaining a life from the discovery of an unknown number of gold mines and the reward received by uncovering all the safe covered cells would be enough to lead to desirable behaviour from an agent. Still, an additional performance measure could be implemented that rewards an agent for flagging cells it determines as dangerous. This measure in conjunction with an agent that knows the number of cells containing daggers in a D&G map, would allow the agent to use less computationally complex procedures to probe cells once it has determined the positions of all the dangerous cells.

In summary, the performance measures can be given as follows:

Performance measure	Score change	Note
Probing a cell	-1	Should minimise the number of probe calls made
Probing a gold mine	$n-1$	1 gold mine makes up for probing all the cells in map -1. Gives a net score increase.
Probing a dagger	$-(n-1)$	Opposite effect of gold mine
Flagging a dangerous cell	$+1$	Desirable – can lead to reduction in heavy computation
Flagging a none-dangerous cell or reflagging a flagged cell	-1000	Extremely undesirable
Winning the game by uncovering all safe cells	$d \times (n-1)$	Gains a score that is equivalent but opposite to probing all the daggers

Table 1 - agent performance measures for D&G game

In terms of actuators, the agent should have the abilities to probe and flag (stone) a cell on the grid. These actuators are implemented in the form of methods that call cell probe and flag methods within the game class. As for sensing, the agent is required to sense the:

- number of adjacent daggers to any cell
- number of daggers in a map
- state of the game (i.e. won/lost/on-going); and
- positions and values of all covered, as well as the locations of uncovered cells

3.0 Implementation

3.1 Game class

The first step with regards to implementation required a robust game class to be developed to function as the environment on which the agents act. A design decision to form complete separation between the game implementation and the agent implementation was taken. This enables the game class to be utilised for more purposes – for example, building a GUI. The game class as designed utilises a single constructor that takes a D&G map in the form of a 2D String array as an input. From this original map, a secondary map is made (named "currentMap") which stores the game state at any step. This secondary map is the map that is accessible to any agent during run-time. The decision to implement this secondary map within the game class rather than the agent classes was taken to ensure that the game class supported many use cases. If the opposite decision was to be taken, the game class would require any client class to implement some form of data structure to keep track of the uncovered cells and their locations - increasing the complexity of using this class's objects.

To implement a probing method capable of auto revealing safe cells, a recursive technique was used. This method follows an algorithm whereby if the cell probed did not contain a gold mine ("g") nor a zero ("0") indicating adjacent cells contain daggers, then only the cell probed is revealed. However, in the case where a cell **is** either of the former, then the chosen cell is probed as well as all of its adjacent cells in a recursive manner until a boundary of numbered cells (numbers 1-8) is formed. Once the method reaches completion, a list of Cell objects (a bespoke Object type created in conjunction with the game class) is returned to the caller which contains the coordinate data and contents of the cells uncovered during a probe call. This appears to be the most efficient way to auto-probe all adjacent non-dangerous cells and update the caller about the cells revealed. Still, to improve efficiency even more, the method was designed to also update fields such as the number of lives left, the number of covered cells remaining, and the number of gold mines collected as the cells are uncovered.

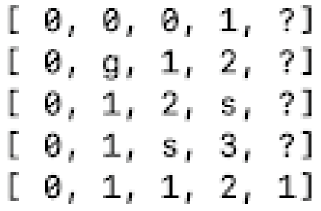
Note: the return list of Cell objects passed back by the probe method serves only as a form of memoisation in this implementation, but it can be useful for many use cases where the data of the uncovered cells are needed. In this implementation, the list allows the agents developed to quickly update their list of covered cells by removing all uncovered cells from their local lists of covered cells.

The next major function implemented was the game state determination method. This method was required to take in two factors into account when determining the game state. The first factor is of course the number of lives left, while the second is the number of none-dangerous covered cells remaining. Games are won only if the number of non-dangerous covered cells remaining is 0 and the number of lives is greater than 0.

3.2 Agent1 class

The first Agent implemented to solve D&G maps can be found in the Agent1 class. This agent contains a single constructor that takes a Game Object as input. Using the given game Object, the agent can begin solving the game by acting upon this Object. As per the spec, Agent1 utilises two separate strategies to solve a given D&G world.

The first strategy utilised by the agent is the SPS, which, attempts to deduce whether an adjacent cell to any uncovered cell contains a dagger (or not) by calculating the difference between the number of threats around the uncovered cell, and the number of total threats indicated by the cell itself. For example, in Figure 1, the uncovered cell at the bottom left of the grid (Cell 4, 4) indicates the there is a single dagger around it. Given that there is only a single covered cell around this cell, then it can be deduced that a dagger must be contained in cell (4, 3). Similarly, the SPS can work in reverse by uncovering an adjacent cell if the number of daggers marked around a cell equals the number of threats indicated by the cell itself. In this case, the agent can deduce that any covered cell around the cell in question must be safe, and as a result will proceed to probe all of the remaining adjacent cells.



```
[ 0, 0, 0, 1, ?]
[ 0, g, 1, 2, ?]
[ 0, 1, 2, s, ?]
[ 0, 1, s, 3, ?]
[ 0, 1, 1, 2, 1]
```

Figure 1 - SPS example

To implement this algorithm in practice, a do-while loop (containing the SPS logic) was utilised to keep carrying out this technique until no changes can be made. This looping strategy is necessary as any changes made, could lead to further cells being solved as a result of a previous solution.
Within the do-while loop, the agent iterates over each cell in the grid and checks the number of covered cells around each uncovered “clue giving” cell (a cell that contains a number from 1-8). If the agent detects that there are more than 0 covered cells around a clue giving cell, then it checks for two conditions, which if met, lead the agent to either uncover or flag all the remaining covered adjacent cells. The condition that leads to the former action is simply to check if the number of daggers as indicated by the clue equals the number of daggers already marked around the clue giving cell; whereas, the latter condition is met if the number of unmarked adjacent cells is equivalent to the clue number minus the number of daggers already marked. The following equations summarise this logic.

$$Uncover\ all\ \leftarrow if\ clue\ \# == \#\ of\ daggers\ already\ marked$$

$$flag\ all\ \leftarrow if\ \#\ covered\ cells == clue\ \# - \#\ of\ daggers\ already\ marked$$

In hindsight, the algorithm implemented is not the most efficient as it cycles through the whole grid after every change. A better approach would have been to maintain a list of uncovered cells that still have covered adjacent cells and iterate over this much smaller list instead of the whole grid. This can be a source of future improvement to the algorithm implemented.

The second strategy implemented allows the agent to probe a covered cell at random. This strategy is intended to be used as a crutch when the SPS reaches a deadlock and is unable to make any more informed changes. To implement this strategy, the agent picks a random index from a list of maintained covered cell Objects – the cell picked is then probed.

The above strategies are utilised by the agent until either the game state changes as determined by the game class, or the number of daggers flagged equals the number of dagger contained in the map to begin with. When the latter condition is met, the agent simply probes all the remaining covered cells before exiting. This allows the agent to reduce the number of SPS computations performed improving its overall efficiency.

3.3 Agent2 class

Agent 2 is an extension of Agent 1 which is capable of solving more difficult D&G worlds and even worlds that Agent1 can usually successfully solve, with the exception that it is capable of relying less on random probing by switching to a strategy that utilises propositional logic to make inferences about the positions of daggers in covered cells using multiple clues at a time.

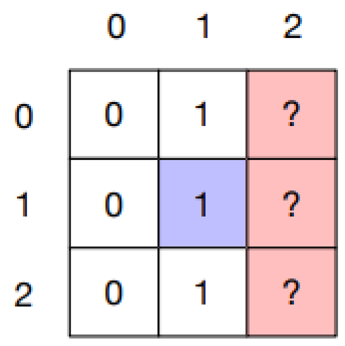


Figure 2 - example situation when logic is required over SPS (Toniolo, 2018)

Consider the example shown in Figure 2. A SPS approach alone would not be able to determine which of the red cells contains a dagger. Yet, in such a situation there is no real need to rely on random probing as it is clear when utilising the information provided by all the clue giving, that there is only one position (2, 1) where a dagger must be to satisfy the constraints provided by the cells in the middle column.

To utilise information obtained from multiple cells (i.e. the knowledge the agent gathers about the world) to make inferences through satisfiability tests, two libraries (LogicNG and ANTLR) were utilised to parse and convert a logical formula (given as a String of sentences about the world) to conjunctive normal form (CNF). From CNF, a third library (SAT4J) was then utilised to test for satisfiability of a particular condition. For example, to test if $KB \rightarrow D2{,}1$ (knowledge base entail a dagger in 2, 1) as shown in Figure 2, the test proposed would be written as:

(~2_0 & 2_1 | 2_0 & ~2_1) & (2_0 & ~2_1& ~2_2 | ~2_0 & 2_1& ~2_2 | ~2_0 & ~2_1& 2_2)

& (~2_1 & 2_2 | 2_1 & ~2_2) & ~2_1

Where: a number such as 2_1 means dagger in cell (2, 1) and ~2_1 means no dagger in (2, 1), | means logical or, and & means logical and. The green part of the formula in this case in the knowledge base, and the red part is the cell and condition being tested for entailment.

The implementation of the process of converting a logical formula to CNF, then converting the resultant literals to DIMACS to be passed into the SAT4J solver can be found in the Solver class. The only point to mention regarding this process is that the SAT4J library required arrays of integers as inputs reflecting the clauses of literals in the knowledge base. To make the conversion from String clauses to integer array clauses, Hash-Maps were used to translate each literal and it's conjugate into positive and negative integers. Another point to mention is that the SAT solver required the user to set the maximum number of variables in each clause before the arrays of literals (the clauses) are passed into the solver. A method was written in the solver class that provides the size of the largest clause. However, this method was not used, and instead the number of atoms in the formula was used to set the maximum size of clauses. This choice was made to reduce computational complexity. Since the number of literals in a clause cannot be greater than the number of atomic formulas in a clause, the number of atoms was a safe option to use.

Once the solver class was constructed and tested, the focus shifted to developing an algorithm that can construct an appropriate knowledge base given a certain condition of the world. The algorithm which can be found in the getKBFormula method in the Agent2 class begins by iterating over every cell in the grid and checking which cells are

1. Uncovered
2. Clue giving; and
3. Have covered adjacent cells

Once a cell with the above characteristics is found, the list of its covered adjacent cells is passed to a method called getSubFormula which returns a String of the possible combinations of dagger and no dagger containing cells in the format shown in the prior formula. This process is repeated for all the cells that meet the above three criteria to construct the full knowledge base formula. The logicProbe method then iterates over each cell which an inference can be made about (red cell in Figure 2 for example) and checks which of the cells can be shown to contain a dagger [1]. Once a dagger is placed, the Agent returns to using the SPS (since SPS is less computationally expensive) to solve the level until it requires the logical approach once again. Note, Agent2 only resorts to random probing when neither the SPS nor the logical probing strategy can make an inference about the world.

[1] A cell can be proven to contain a dagger by satisfying that the knowledge base does not entail no-dagger in a specific cell.

As with Agent1, a suggested improvement to efficiency can be made by maintaining a list of uncovered cells that have covered adjacent cells. This would reduce the number of cells iterated over by the getKBFormula, logicProbe, and spsProbe methods.

3.4 FormulaBuilder class

To construct all the combinations of adjacent covered cells that contain daggers and those that do not, a formula builder class was made. This class contains a constructor that accepts an ArrayList of cell objects (the covered cells adjacent to a clue giving cell). To get the formula, the getFormula method is then used and takes an integer parameter as an input representing the number of daggers (i.e. combinations) contained within the list of covered adjacent cells. This class then utilises a recursive method to return all the possible combinations of dagger locations in the form of a logical formula.

This class can easily be adapted to return combinations of any objects in an ArrayList, and if further work was taken, this would be carried out.

3.5 Launcher classes

The final parts of the implementation are the two main method classes that allow either agent to be used to solve any of the maps contained in the enum found in the World.java file.

To run Agent1 the following command needs to be entered on the terminal from within the src directory:

$$java\ Launcher1\ < world\_name > \quad e.g.java\ Launcher1\ EASY1$$

Similarly, agent 2 can be run using the command:

$$Java - cp\ .:../libraries/antlr - 4.7.1 - complete.jar:../libraries/logicng - 1.4.0.jar:../libraries/sat4j - pb.jar\ Launcher2\ < world\_name >$$

Note: jars are provided for both agents.

4.0 Testing

4.1 Game implementation

The Game class clearly needed to be tested extensively to ensure that it operates as expected. Without a correctly functioning game class, no agent regardless of the correctness of its implementation would be able to solve the game unless the agent could learn the rules of the broken game.

An automated test was thus developed to check the game class functionality. The test can be found in the GameTester class, and can be initiated for any world by passing a world name as a command-line argument when running the test.

$$java\ GameTester\ < world\_name >\quad e.g. java\ GameTester\ EASY1$$

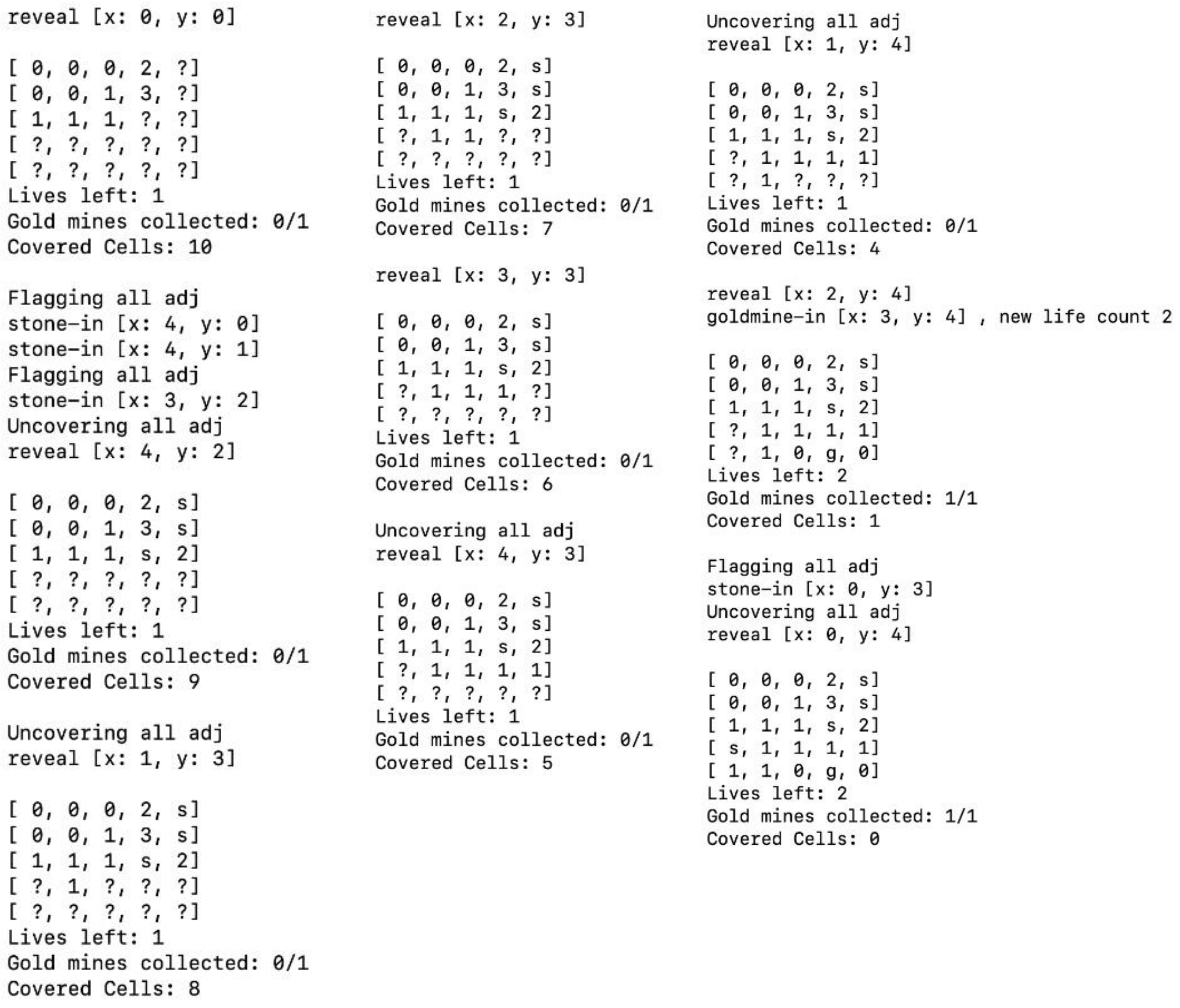
The automated tester checks the implementations of the following:

- Initial map generation showing hidden cells.
- The initial stats including the lives remaining, gold mines collected, and covered cells expected.
- The resultant behaviour of a probe call to a 0 or g cell. Expects an iterative uncovering of all adjacent cells that have no threats around them leaving a boundary of cells that contain threats. This is checked by ensuring that every newly uncovered cell is a direct neighbour to a "0" or "g" cell, and each "0" or "g" cell is itself a neighbour to another "0" or "g". Furthermore, checks are made to ensure that each covered cell is not adjacent to a "0" or "g" cell.
- The update functionality of the game stats (given the previous probe call). The lives, gold mines collected, and covered cell counters are checked. This check is performed by counting the cells in the displayable map and comparing them with the fields in the game class.
- The resultant behaviour of a probe call to a dagger cell. The results expected are that only the cell probed is unveiled, and the lives count decrements while the cells covered count remains the same.
- The resultant behaviour of a flag (stone) call. Check performed by counting the number of cells flagged. If the count is 1 and the flagged cell coordinates are as expected, then this test is passed.

4.2 Agent1

Agent1 was tested manually to ensure its functionalities worked as expected. For example, print statements were used to ensure that the coordinates the agent expects to probe/flag match the coordinates of the cell actually probed/flagged.

The most challenging part to test was to prove that the single point strategy worked as expected. To test for this the SPS solution to map "EASY3" was carried out by hand and compared to the results shown in Figure 3. The figure shows a printout of the map after each probe operation performed by the agent. The figure also prints out each operation performed by the agent in sequence. For example, after the first probe call to (0, 0), the SPS leads the agent to flag all covered cells adjacent to (3, 0) and then repeats the same to all cells adjacent to (2, 1). This map proves that the SPS works as expected, however, it does not show how the agent behaves once the SPS can no longer lead to any solutions. Figure 4 shows a situation where the SPS is no longer helpful, and how the agent resorts to using random probing to proceed. Finally, further checks were then carried out to ensure that the list of remaining covered cells the agent maintains is accurate. However, the code is itself self-checking in this respect as each probed cell is removed from the list of covered cells. If the cell did not exist, then an exception would be thrown.

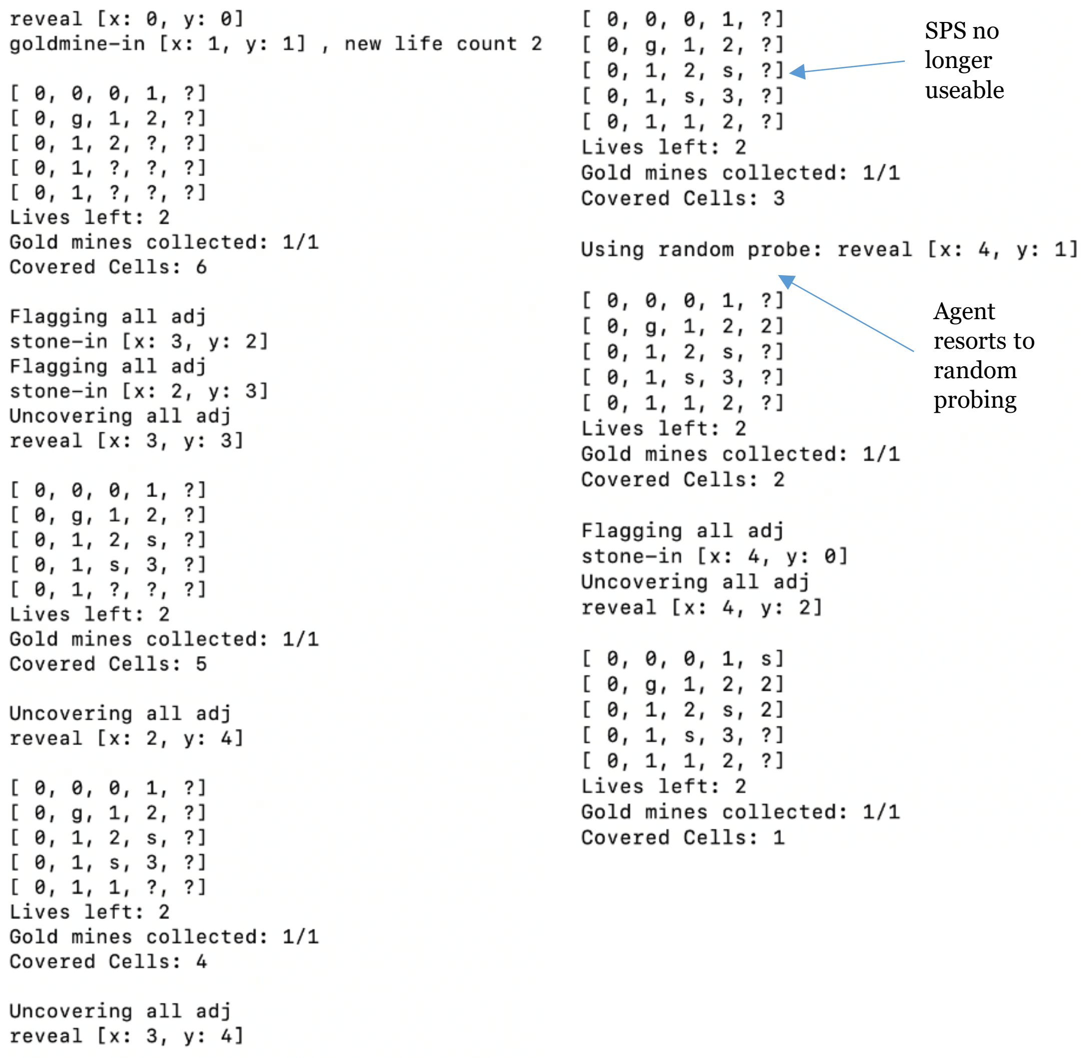


```
reveal [x: 0, y: 0]

[ 0, 0, 0, 2, ?]
[ 0, 0, 1, 3, ?]
[ 1, 1, 1, ?, ?]
[ ?, ?, ?, ?, ?]
[ ?, ?, ?, ?, ?]
Lives left: 1
Gold mines collected: 0/1
Covered Cells: 10

Flagging all adj
stone-in [x: 4, y: 0]
stone-in [x: 4, y: 1]
Flagging all adj
stone-in [x: 3, y: 2]
Uncovering all adj
reveal [x: 4, y: 2]

[ 0, 0, 0, 2, s]
[ 0, 0, 1, 3, s]
[ 1, 1, 1, s, 2]
[ ?, ?, ?, ?, ?]
[ ?, ?, ?, ?, ?]
Lives left: 1
Gold mines collected: 0/1
Covered Cells: 9

Uncovering all adj
reveal [x: 1, y: 3]

[ 0, 0, 0, 2, s]
[ 0, 0, 1, 3, s]
[ 1, 1, 1, s, 2]
[ ?, 1, ?, ?, ?]
[ ?, ?, ?, ?, ?]
Lives left: 1
Gold mines collected: 0/1
Covered Cells: 8
```

```
reveal [x: 2, y: 3]

[ 0, 0, 0, 2, s]
[ 0, 0, 1, 3, s]
[ 1, 1, 1, s, 2]
[ ?, 1, 1, ?, ?]
[ ?, ?, ?, ?, ?]
Lives left: 1
Gold mines collected: 0/1
Covered Cells: 7

reveal [x: 3, y: 3]

[ 0, 0, 0, 2, s]
[ 0, 0, 1, 3, s]
[ 1, 1, 1, s, 2]
[ ?, 1, 1, 1, ?]
[ ?, ?, ?, ?, ?]
Lives left: 1
Gold mines collected: 0/1
Covered Cells: 6

Uncovering all adj
reveal [x: 4, y: 3]

[ 0, 0, 0, 2, s]
[ 0, 0, 1, 3, s]
[ 1, 1, 1, s, 2]
[ ?, 1, 1, 1, 1]
[ ?, ?, ?, ?, ?]
Lives left: 1
Gold mines collected: 0/1
Covered Cells: 5
```

```
Uncovering all adj
reveal [x: 1, y: 4]

[ 0, 0, 0, 2, s]
[ 0, 0, 1, 3, s]
[ 1, 1, 1, s, 2]
[ ?, 1, 1, 1, 1]
[ ?, 1, ?, ?, ?]
Lives left: 1
Gold mines collected: 0/1
Covered Cells: 4

reveal [x: 2, y: 4]
goldmine-in [x: 3, y: 4] , new life count 2

[ 0, 0, 0, 2, s]
[ 0, 0, 1, 3, s]
[ 1, 1, 1, s, 2]
[ ?, 1, 1, 1, 1]
[ ?, 1, 0, g, 0]
Lives left: 2
Gold mines collected: 1/1
Covered Cells: 1

Flagging all adj
stone-in [x: 0, y: 3]
Uncovering all adj
reveal [x: 0, y: 4]

[ 0, 0, 0, 2, s]
[ 0, 0, 1, 3, s]
[ 1, 1, 1, s, 2]
[ s, 1, 1, 1, 1]
[ 1, 1, 0, g, 0]
Lives left: 2
Gold mines collected: 1/1
Covered Cells: 0
```

Figure 3 - map EASY3 SPS solution results



```
reveal [x: 0, y: 0]
goldmine-in [x: 1, y: 1] , new life count 2

[ 0, 0, 0, 1, ?]
[ 0, g, 1, 2, ?]
[ 0, 1, 2, ?, ?]
[ 0, 1, ?, ?, ?]
[ 0, 1, ?, ?, ?]
Lives left: 2
Gold mines collected: 1/1
Covered Cells: 6

Flagging all adj
stone-in [x: 3, y: 2]
Flagging all adj
stone-in [x: 2, y: 3]
Uncovering all adj
reveal [x: 3, y: 3]

[ 0, 0, 0, 1, ?]
[ 0, g, 1, 2, ?]
[ 0, 1, 2, s, ?]
[ 0, 1, s, 3, ?]
[ 0, 1, ?, ?, ?]
Lives left: 2
Gold mines collected: 1/1
Covered Cells: 5

Uncovering all adj
reveal [x: 2, y: 4]

[ 0, 0, 0, 1, ?]
[ 0, g, 1, 2, ?]
[ 0, 1, 2, s, ?]
[ 0, 1, s, 3, ?]
[ 0, 1, 1, ?, ?]
Lives left: 2
Gold mines collected: 1/1
Covered Cells: 4

Uncovering all adj
reveal [x: 3, y: 4]
```

```
[ 0, 0, 0, 1, ?]
[ 0, g, 1, 2, ?]
[ 0, 1, 2, s, ?]
[ 0, 1, s, 3, ?]
[ 0, 1, 1, 2, ?]
Lives left: 2
Gold mines collected: 1/1
Covered Cells: 3

Using random probe: reveal [x: 4, y: 1]

[ 0, 0, 0, 1, ?]
[ 0, g, 1, 2, 2]
[ 0, 1, 2, s, ?]
[ 0, 1, s, 3, ?]
[ 0, 1, 1, 2, ?]
Lives left: 2
Gold mines collected: 1/1
Covered Cells: 2

Flagging all adj
stone-in [x: 4, y: 0]
Uncovering all adj
reveal [x: 4, y: 2]

[ 0, 0, 0, 1, s]
[ 0, g, 1, 2, 2]
[ 0, 1, 2, s, 2]
[ 0, 1, s, 3, ?]
[ 0, 1, 1, 2, ?]
Lives left: 2
Gold mines collected: 1/1
Covered Cells: 1
```

Figure 4- map EASY1 SPS and Random probing combination

4.5 Agent2

Agent2 like Agent1 was also tested through inspection. However, due to the complexity of this agent, different parts were tested individually first.

Solver class

The solver class was tested separately from the agent by creating a separate main class which creates solver objects and passes a formula in the form of a string to test for satisfiability. The formula at this point was written out by hand for certain maps where the answer was known beforehand. For example, the formula shown above for Figure 2 returns an unsatisfiable result indicating that a dagger does indeed exist in cell (2, 1). More expressions were written and tested to confirm that the solver works as expected.

Formula Builder

The FormulaBuilder class was tested next. It is vital that the formula builder class produces all the possible combinations of dagger locations to provide all the constraints necessary for the solver to produce the correct results. To test this class, a tester class (named Combination.java) was written. In this class, 5 dummy cells are created and added to an ArrayList named coveredCells which is passed into the FormulaBuilder object. A for loop was then used to increase the number of daggers within the 5 cells to see if all the combinations are successfully found. The following figure shows the results of this test which proves that the combinations program works as expected.

```
Number of daggers: 1
Formula: (1 & ~2 & ~3 & ~4 & ~5| 2 & ~1 & ~3 & ~4 & ~5| 3 & ~1 & ~2 & ~4 & ~5| 4 & ~1 & ~2 & ~3 & ~5| 5 & ~1 & ~2 &
 ~3 & ~4)

Number of daggers: 2
Formula: (1 & 2 & ~3 & ~4 & ~5| 1 & 3 & ~2 & ~4 & ~5| 1 & 4 & ~2 & ~3 & ~5| 1 & 5 & ~2 & ~3 & ~4| 2 & 3 & ~1 & ~4 &
 ~5| 2 & 4 & ~1 & ~3 & ~5| 2 & 5 & ~1 & ~3 & ~4| 3 & 4 & ~1 & ~2 & ~5| 3 & 5 & ~1 & ~2 & ~4| 4 & 5 & ~1 & ~2 & ~3)

Number of daggers: 3
Formula: (1 & 2 & 3 & ~4 & ~5| 1 & 2 & 4 & ~3 & ~5| 1 & 2 & 5 & ~3 & ~4| 1 & 3 & 4 & ~2 & ~5| 1 & 3 & 5 & ~2 & ~4|
1 & 4 & 5 & ~2 & ~3| 2 & 3 & 4 & ~1 & ~5| 2 & 3 & 5 & ~1 & ~4| 2 & 4 & 5 & ~1 & ~3| 3 & 4 & 5 & ~1 & ~2)

Number of daggers: 4
Formula: (1 & 2 & 3 & 4 & ~5| 1 & 2 & 3 & 5 & ~4| 1 & 2 & 4 & 5 & ~3| 1 & 3 & 4 & 5 & ~2| 2 & 3 & 4 & 5 & ~1)
```

Figure 5 - results of combinations testing

Logic probing strategy

The final tests carried out involved testing the overall functionality of the agent on maps that cannot be solved by SPS alone, but, can be solved through a combination of SPS and logical reasoning. The two maps and their results are shown in figures 6 – 9.

	0	1	2	3
0	0	0	2	🗡️
1	1	1	4	🗡️
2	1	🗡️	3	🗡️
3	1	1	2	1

Figure 6 – agent 2 test map 1

	0	1	2	3	4
0	0	0	1	2	2
1	0	1	3	⚠️	⚠️
2	0	2	⚠️	⚠️	3
3	0	2	⚠️	3	1
4	0	1	1	1	0

Figure 7 - agent 2 test map 2

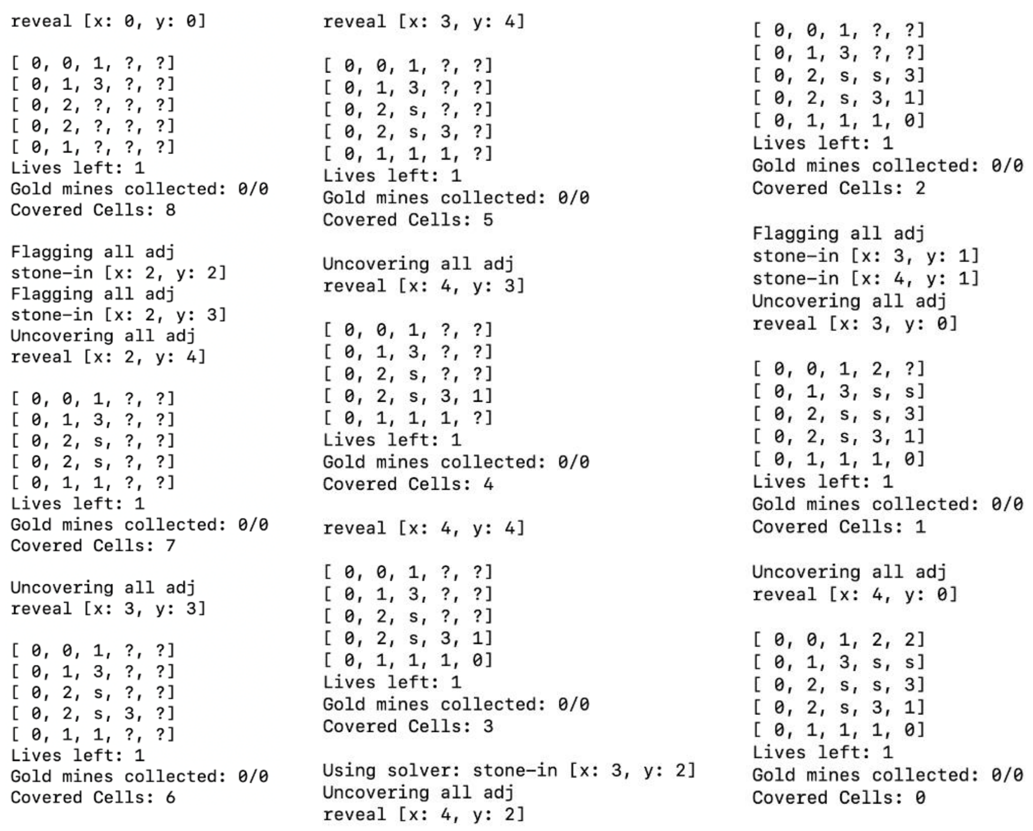
```
reveal [x: 0, y: 0]

[ 0, 0, 2, ?]
[ 1, 1, 4, ?]
[ ?, ?, ?, ?]
[ ?, ?, ?, ?]
Lives left: 1
Gold mines collected: 0/0
Covered Cells: 6

Flagging all adj
stone-in [x: 3, y: 0]
stone-in [x: 3, y: 1]
Using solver: stone-in [x: 1, y: 2]
Uncovering all adj
reveal [x: 0, y: 2]

[ 0, 0, 2, s]
[ 1, 1, 4, s]
[ 1, s, ?, ?]
[ ?, ?, ?, ?]
Lives left: 1
Gold mines collected: 0/0
Covered Cells: 5

Uncovering all adj
reveal [x: 2, y: 2]

[ 0, 0, 2, s]
[ 1, 1, 4, s]
[ 1, s, 3, ?]
[ ?, ?, ?, ?]
Lives left: 1
Gold mines collected: 0/0
Covered Cells: 4

Flagging all adj
stone-in [x: 3, y: 2]
Uncovering all adj
reveal [x: 0, y: 3]
```

Agent utilises the solver to deduce a dagger exists in 1, 2

```
[ 0, 0, 2, s]
[ 1, 1, 4, s]
[ 1, s, 3, s]
[ 1, ?, ?, ?]
Lives left: 1
Gold mines collected: 0/0
Covered Cells: 3

reveal [x: 1, y: 3]

[ 0, 0, 2, s]
[ 1, 1, 4, s]
[ 1, s, 3, s]
[ 1, 1, ?, ?]
Lives left: 1
Gold mines collected: 0/0
Covered Cells: 2

Uncovering all adj
reveal [x: 2, y: 3]

[ 0, 0, 2, s]
[ 1, 1, 4, s]
[ 1, s, 3, s]
[ 1, 1, 2, ?]
Lives left: 1
Gold mines collected: 0/0
Covered Cells: 1

reveal [x: 3, y: 3]

[ 0, 0, 2, s]
[ 1, 1, 4, s]
[ 1, s, 3, s]
[ 1, 1, 2, 1]
Lives left: 1
Gold mines co
Covered Cells
```

0 random probes

Figure 8 - Results of agent 2 test map 1



```
reveal [x: 0, y: 0]

[ 0, 0, 1, ?, ?]
[ 0, 1, 3, ?, ?]
[ 0, 2, ?, ?, ?]
[ 0, 2, ?, ?, ?]
[ 0, 1, ?, ?, ?]
Lives left: 1
Gold mines collected: 0/0
Covered Cells: 8

Flagging all adj
stone-in [x: 2, y: 2]
Flagging all adj
stone-in [x: 2, y: 3]
Uncovering all adj
reveal [x: 2, y: 4]

[ 0, 0, 1, ?, ?]
[ 0, 1, 3, ?, ?]
[ 0, 2, s, ?, ?]
[ 0, 2, s, ?, ?]
[ 0, 1, 1, ?, ?]
Lives left: 1
Gold mines collected: 0/0
Covered Cells: 7

Uncovering all adj
reveal [x: 3, y: 3]

[ 0, 0, 1, ?, ?]
[ 0, 1, 3, ?, ?]
[ 0, 2, s, ?, ?]
[ 0, 2, s, 3, ?]
[ 0, 1, 1, ?, ?]
Lives left: 1
Gold mines collected: 0/0
Covered Cells: 6
```

```
reveal [x: 3, y: 4]

[ 0, 0, 1, ?, ?]
[ 0, 1, 3, ?, ?]
[ 0, 2, s, ?, ?]
[ 0, 2, s, 3, ?]
[ 0, 1, 1, 1, ?]
Lives left: 1
Gold mines collected: 0/0
Covered Cells: 5

Uncovering all adj
reveal [x: 4, y: 3]

[ 0, 0, 1, ?, ?]
[ 0, 1, 3, ?, ?]
[ 0, 2, s, ?, ?]
[ 0, 2, s, 3, 1]
[ 0, 1, 1, 1, ?]
Lives left: 1
Gold mines collected: 0/0
Covered Cells: 4

reveal [x: 4, y: 4]

[ 0, 0, 1, ?, ?]
[ 0, 1, 3, ?, ?]
[ 0, 2, s, ?, ?]
[ 0, 2, s, 3, 1]
[ 0, 1, 1, 1, 0]
Lives left: 1
Gold mines collected: 0/0
Covered Cells: 3

Using solver: stone-in [x: 3, y: 2]
Uncovering all adj
reveal [x: 4, y: 2]
```

```
[ 0, 0, 1, ?, ?]
[ 0, 1, 3, ?, ?]
[ 0, 2, s, s, 3]
[ 0, 2, s, 3, 1]
[ 0, 1, 1, 1, 0]
Lives left: 1
Gold mines collected: 0/0
Covered Cells: 2

Flagging all adj
stone-in [x: 3, y: 1]
stone-in [x: 4, y: 1]
Uncovering all adj
reveal [x: 3, y: 0]

[ 0, 0, 1, 2, ?]
[ 0, 1, 3, s, s]
[ 0, 2, s, s, 3]
[ 0, 2, s, 3, 1]
[ 0, 1, 1, 1, 0]
Lives left: 1
Gold mines collected: 0/0
Covered Cells: 1

Uncovering all adj
reveal [x: 4, y: 0]

[ 0, 0, 1, 2, 2]
[ 0, 1, 3, s, s]
[ 0, 2, s, s, 3]
[ 0, 2, s, 3, 1]
[ 0, 1, 1, 1, 0]
Lives left: 1
Gold mines collected: 0/0
Covered Cells: 0
```

Figure 9 - results of agent 2 test map 2

5.0 Evaluation

To evaluate the performance of both agents, results were obtained for each agent on all 30 maps. Data recorded for both agents included the number of random probes made, SPS cycles (complete iterations through the game grid), and win rate achieved. Additionally, the number of satisfiability tests carried out and satisfiability solutions that resulted from those tests were also recorded. These latter parameters allow the usefulness of the logic-probe strategy to be scrutinised.

Map		Random probes					Av:	SPS cycles					Av:	Win ratio
EASY	1	1	2	2	1	1	1.4	7	7	7	7	7	7	5/5
	2	6	2	4	1	4	3.4	10	6	8	2	7	6.6	4/5
	3	0					0	3					3	5/5
	4	0					0	3					3	5/5
	5	3	6	3	4	2	3.6	8	10	7	8	7	8	5/5
	6	0					0	4					4	5/5
	7	4	2	3	2	1	2.4	7	5	7	6	4	5.8	5/5
	8	2	2	5	1	1	2.2	5	6	5	5	4	5	5/5
	9	0					0	3					3	5/5
	10	0					0	3					3	5/5
MEDIUM	1	0					0	5					5	5/5
	2	1	1	1	3	3	1.8	9	9	9	11	11	9.8	5/5
	3	0					0	3					3	5/5
	4	2	7	3	1	1	2.8	7	12	8	6	6	7.8	5/5
	5	4	3	4	4	9	4.8	9	8	12	10	17	11.2	5/5
	6	1	3	3	3	2	2.4	6	9	10	9	10	8.8	5/5
	7	2	2	3	2	3	2.4	5	5	6	5	6	5.4	5/5
	8	1	1	2	1	3	1.6	7	7	8	7	5	6.8	5/5

	9	1	1	2	1	1	1.2	7	8	7	8	7	7.4	5/5
	10	3	3	1	2	1	2	10	7	6	6	5	6.8	5/5
HARD	1	1	6	2	4	6	3.8	3	12	9	14	8	9.2	3/5
	2	6	3	2	1	2	2.8	19	12	16	2	3	10.4	3/5
	3	8	5	5	11	2	6.2	23	19	18	24	7	18.2	4/5
	4			0			0			5			5	5/5
	5	5	6	10	9	9	7.8	21	25	21	19	19	21	4/5
	6	6	2	1	4	3	3.2	12	2	10	11	13	9.6	4/5
	7	3	2	12	5	2	4.8	19	19	28	18	5	17.8	3/5
	8	5	7	4	9	5	6	13	16	17	17	15	15.6	3/5
	9	8	7	1	22	4	8.4	20	20	4	24	7	15	3/5
	10	7	3	10	8	6	6.8	15	15	17	20	15	16.4	4/5
Average							2.7						8.6	90%
Average green maps							4.7							

Table 2 - Agent 1 results

Map		Random probes					Av	SPS cycles					Av	Win ratio
EASY	1	1	1	1	1	1	1	4	4	7	4	6	5	5/5
	2	2	1	2	3	4	2.4	8	5	3	4	8	5.6	3/5
	3			0			0			3			3	5/5
	4			0			0			3			3	5/5
	5	3	3	3	2	3	2.8	4	8	9	3	6	6	3/5
	6			0			0			4			4	5/5
	7			0			0			4			4	5/5
	8			0			0			4			4	5/5
	9			0			0			3			3	5/5
	10			0			0			3			3	5/5
MEDIUM	1			0			0			5			5	5/5
	2	1	1	3	3	1	1.8	9	9	10	11	10	9.8	5/5
	3			0			0			3			3	5/5
	4	5	1	1	2	4	2.6	10	6	6	7	9		5/5
	5			0			0			7			7	5/5
	6	4	1	1	4	4	2.8	11	9	9	10	9	8.2	5/5
	7	1	2	2	3	1	1.8	4	6	5	6	4	5	5/5
	8	3	2	2	1	2	2	6	9	9	8	9	8.2	4/5
	9	1	2	1	2	1	1.4	7	7	7	8	7	7.2	5/5
	10			0			0			5			5	5/5
HARD	1	3	1	3	1	1	1.8	11	4	14	12	4	9	3/5
	2	4	2	3	1	3	2.6	6	3	18	15	13	11	3/5
	3			0			0			11			11	5/5
	4			0			0			5			5	5/5
	5			0			0			14			14	5/5
	6	5	1	1	2	2	2.2	15	1	11	11	2	8	3/5
	7	1	1	1	1	1	1	17	18	17	18	17	17.4	5/5
	8	5	7	2	2	7	4.6	19	8	3	3	20	10.6	2/5
	9	2	2	2	3	2	2.2	13	19	18	18	20	17.6	4/5
	10	4	3	6	2	1	3.2	20	15	17	18	2	14.4	4/5
Average							1.2						7.5	89%
Average green maps							0.84							

Table 3 - Agent 2 results

Map		Satisfiability tests					Av	Satisfiability solutions					Av
EASY	1	5	5	5	5	5	5	0	0	0	0	0	0
	2	22	7	18	25	38	22	1	0	0	0	0	0.2
	3			0			0			0			0

	4	0					0	0					0
	5	23	28	32	11	26	24	0	1	1	0	0	0.4
	6	0					0	0					0
	7	2					2	1					1
	8	1					1	1					1
	9	0					0	0					0
	10	0					0	0					0
MEDIUM	1	0					0	0					0
	2	6	6	21	22	9	12.8	0	0	0	0	1	0.2
	3	0					0	0					0
	4	97	7	7	16	62	37.8	0	0	0	0	0	0
	5	2					2	1					1
	6	57	6	6	60	54	36.6	0	0	0	0	0	0
	7	4	9	9	13	4	7.8	0	0	0	0	0	0
	8	50	33	32	15	32	32.4	1	1	1	1	1	1
	9	3	6	3	6	3	4.2	0	0	0	0	0	0
	10	3					3	1					1
HARD	1	70	20	58	20	20	37.6	1	1	1	1	1	1
	2	66	35	71	17	69	51.6	0	0	1	1	1	0.6
	3	1					1	1					1
	4	0					0	0					0
	5	26					26	4					4
	6	105	7	7	22	16	31.4	0	0	0	0	0	0
	7	16	16	16	16	16	16	2	2	2	2	2	2
	8	115	188	24	24	243	118.8	3	0	0	0	3	1.2
	9	49	35	35	68	44	46.2	2	2	2	2	2	2
	10	76	45	151	35	7	62.8	1	0	1	1	0	0.6
Average (All maps)							19.4						0.6
Average (maps benefiting from SAT tests)							26.5						1.21
Average (maps not benefiting from SAT tests							20.8						0.2

Table 4 - Agent 2 extended results

Note: Rows highlighted in green mark the maps that benefited from the logical probe strategy.

6.0 Conclusion

The results presented in the evaluation section may lead to the false assumption that both agents performed comparably for all the maps if the win-rate was compared. In actuality, it should be noted that while the win-rates were very similar, the numbers would be different on the maps highlighted in green if a more statistically significant number of runs were recorded. This can be deduced because on average, Agent2 reduced the number of random probes made on those maps by around 4 probes. This observation leads to a more sound conclusion that agent 2 in fact had a major impact solving the more difficult maps with more reliability as it reduced the chances of probing a dagger on easy and medium maps by around 15% and on difficult maps by around 50%.

Nevertheless, Agent2 was only useful for only a set of 10 maps out of the 30 maps. On the maps that the additional probing strategy did not lead to any benefits, a significant computational cost was incurred as more than 20 satisfiability tests were conducted in each. This information combined with the fact that on average 26 computationally intensive satisfiability tests were required per satisfiability solution should make it clear to any designers that a trade-off between speed and accuracy should be considered when utilising any of the agents developed.

7.0 References

Bayer, K. M., & Snyder, J. (2006). An Interactive Constraints-Based Approach to Minesweeper. *The Twenty-First National Conference on Artificial Intelligence*, (pp. 1933-1934).

Castillo, L. P., & Wrobel, S. (2003). Learning Minesweeper with Multirelational Learning. *IJCAI-03*, (pp. 533-540). Acapulco, Mexico.

David, B. J. (2015). *Algorithmic Approaches to Playing Minesweeper*. Cambridge, Massachusetts: Harvard College.

Kaye, R. (2000). Minesweeper is np-complete. *Mathematical Intelligencer*, 9-15.

Massaioli , R. (2013, January 12). *Solving Minesweeper with Matricies*. Retrieved from wordpress: https://massaioli.wordpress.com/2013/01/12/solving-minesweeper-with-matricies/comment-page-1/

Russell, S., & Norvig, P. (2016). *Artificial Intelligence A Modern Approach* (3rd ed.). Upper Saddle River, NJ, USA: Pearson.

Scott, A., Stege, U., & Rooij, I. V. (2011). Minesweeper May Not Be NP-Complete but Is Hard Nonetheless. *Mathematical Intelligencer*, 5-17.

Studholme, C. (2000). *Minesweeper as a constraints satisfaction problem.*

Toniolo, A. (2018, October 11). CS5011 Logical Agents Part 2 and Assignment. St-Andrews, Fife, Scotland.