CS5011 A2

150006108

Octorber - November 2018

1 Part 1

1.1 PEAS Agent Model

1.1.1 Performance Measure

- Gold = +1 Lives
- Dagger = -1 Lives
- If Lives = 0, game over
- If all cells are either flagged or uncovered, you have won

1.1.2 Environment

- Value of cell represents number of daggers in neighbouring cells
- Finding gold grants you a life and reveals all neighbouring cells
- Finding a "0" reveals all neighbouring cells
- Finding a dagger shall remove a life
- Cells can be flagged if it is believed they contain a dagger

1.1.3 Actuators

- Random Probe
- Single Point Inspection
- Satisfiability Test

1.1.4 Sensors

- Number of daggers in adjacent cells
- Flagged cells
- Covered cells

1.2 Game Infrastructure

The game map is stored as a two dimensional array of Strings. Each string represents a different state.

- "0 - 8" - The number of daggers found in adjacent tiles
- "d" - A cell containing a dagger
- "g" - A cell containing gold
- "D" - A cell that is believed to contain a dagger

1.3 Random Probe and Single Point Agent

When an agent is created, a two dimensional array is instantiated with the same dimensions as the map. Each cell in this new array is filled with an "x". This denotes a covered cell. The array itself represents the agents current knowledge of the map.

1.3.1 Making Moves

As described in the specification, an agents first move must always be to probe [0, 0]. Since this cell never has any daggers adjacent to it, all the neighbouring cells will be revealed.

In my implementation of this, any cells containing either "0" or "g" will have their neighbours also uncovered. This is done in a recursive method, meaning in one move, all safe cells are uncovered. Once this is completed, the random probing can begin.

After the first move, the agent enters a loop that will only terminate once an end game state has been reached, those being that the agent has either ran out of lives every cell in the agents knowledge is flagged or uncovered. These win conditions are checked after each move made by the agent. If neither conditions have been met, the agent can make another move.

1.3.2 Random Probe

The random probe method is pretty straightforward. When the agent enters its solving loop, the *randomProbe* method is called. In this method, a set of coordinates are randomly generated. The respective cell is inspected. If it is covered, the cell is probed and the method returns *true* to signify a valid move has been made. However, if the random coordinates are found to contain a cell that has already been uncovered, the game state would be unaffected by the move, so the method returns false, signifying that no move has been made. If a valid move has been made, the agents updated knowledge is printed.

After a move has been attempted, the end game conditions are checked. If the agents lives equals 0, the game is over. Since the random probing method doesn't involve flagging suspect cells, a check is used to count the amount of covered and revealed dagger cells. If it equals the amount of daggers contained within the map, all remaining covered cells must be daggers, so the game has been won.

1.3.3 Single Point Strategy

The single point strategy is a much more informed approach to solving the game. When the agent enters its solving loop, each cell is inspected individually. If a cell is covered, two checks are made; *AllFreeNeighbours* and *AllMarkedNeighbours*.

The first checks all of the covered cells neighbours. If the cell contains a clue, and the amount of flagged daggers surrounding *that* cell is equivalent to the value of the clue cell, then it is impossible for the cell being inspected to be a dagger, and we can therefore probe the covered cell.

The second checks each of the covered cells neighbours, counting how many covered and flagged cells neighbour it. If the amount of covered neighbouring cells equal the current neighbours clue value *minus* the amount of cells surrounding it that are flagged, we can be certain our current covered cell is a dagger, and so we flag it and move on.

If both of these tests return false, we cannot currently be certain the cell is or isn't a dagger, so we leave it and move on.

If the agent goes through its entire knowledge without making a change, no valid move can be made using the Single Point Strategy, so we must take a chance and randomly probe a cell. Hopefully this newly revealed cell will contain a clue that will enable the SPS to deduce new information regarding the map.

This method of inspection is carried out until one of the completeness conditions is met.

1.3.4 Viability

By using random probing only, it is almost impossible to win. The chances of the agent selecting a cell without a dagger in it decrease through time, as each successful move increases the odds of finding a dagger in the next move, until eventually there will be more dagger cells than non-dagger cells. Also, as the difficulty of the maps increase, the amount of available lives becomes increasingly disproportionate to the amount of daggers within the map, meaning it is even easier to lose.

The introduction of the Single Point Strategy allows the agent to make informed decisions based on its current knowledge of the map. This greatly reduces the need to make random guesses.

When using both methods in the first map (EASY1), after the first move, there are 10 covered cells remaining, 4 of which contain daggers. This means the random probe, on its first move, has a 40% chance of finding a dagger. Now, luckily some gold is revealed in the first move, so the agent has an extra life, but the odds of losing are still enormous.

In comparison, using SPS, the agent doesn't have to resort to using a random probe until there are only 5 covered cells left, 2 of which are daggers. Now, there is the same probability of the agent hitting a dagger as the previous case, however, the agent still has both its lives, and revealing either one of the daggers allows us to deduce the position of the second, meaning it is certain the agent will complete the game.

In summary, the random probe agent would have to make a minimum of 6 random probes, all avoiding a dagger, in order to win, with each probe becoming less and less likely to not hit a dagger. Using SPS, the agent is **guaranteed** to complete the map. There is no contest.

As the difficulty of the maps increase, SPS struggles to scale, as it relies more and more on a random probe revealing relevant information that the agent can utilise to expand its knowledge. For example, using the map HARD1, very little information is revealed in the first dew turns, meaning a random probe is required in a very large, covered expanse. The agent currently has only 1 life, there are 120 covered blocks, 25 of which are daggers. This gives the agent roughly a 20% chance of failing, and even if it doesn't hit a dagger, the chances of the revealed cell containing any currently useful information are even lower. This just means another random probe will be required afterwards, with an increased chance of hitting a dagger.

Although still more likely to complete the map that simply using a random probe, the odds are stacked against the agent. A more sophisticated approach is required.

2 Part 2 - Satisfiability Test

To decrease the amount of random probing required, we can perform a logical check on each covered cell that neighbours an uncovered cell with respect to all over covered cells, and prove whether or not the cell is safe to probe.

A satisfiability check will only be carried out if an SPS loop fails to make a move. Each uncovered cell that neighbours at least one covered cell has a logi-

cal formula built, with the literals of this formula representing the covered cells. The clause of the formula contains a case where only one of the covered cells is a dagger. A clause for each cell being the dagger is generated. These are then ORed together to build the final formula for the uncovered cell.

Each uncovered cell with neighbouring covered cells has a formula like this generated. Each formula gets stored in a list. Once every viable cell has been inspected, all the stored formulas get ANDed together into one single, large formula. This is the KBU.

The library LogicNG is then used to convert the KBU from propositional logic into CNF. Issues were found when using LogicNG to convert larger formulas, requiring the literals used to be named in a specific fashion in order to work around this problem.

With the formula now in CNF, we can convert it to DIMACS, where each literal is represented by a unique integer value, as this is the format SAT4J uses to test satisfiability. The formula is parsed, with each clause being added to another list. Using this list, it is possible to then extract the literals in DIMACS form and add each clause to the SAT4J ISolver.

The final clause added to the formula is the relevant literal for the cell we are trying to prove is safe.

$$SAT(KBU \wedge D(x, y))$$

If the formula is **not** satisfiable, the cell being checked is proven to be safe, and we can therefore probe it.

If all cells are found to satisfy the formula, we cannot be certain that any of them are safe to probe. It is only in this case that the agent should resort to random probing, and both SPS and ATS have failed to deduce a logical move.

2.1 Viability

Controversially, this new agent actually performs *worse* on the map EASY1 than just using SPS. A satisfiability test is ran when the SPS loop fails to make a move, but the revealed cell still fails to help the agent deduce the location of either of the two daggers. Another test is ran on this new knowledge, but it is equal probability that a dagger could be in any of the 4 remaining cells, so a random probe is required. The agent is still certain of completing the map without dying, but the satisfiability test is ultimately redundant.

The same cannot be said for the map HARD1. Where the SPS agent had substantial chance of failure, the ATS agent is guaranteed to succeed. At the point

where the SPS agent would have to use a random probe, the ATS agent runs a satisfiability test. This single test reveals one cell that, consequently, allows the agent to use SPS to solve the entire rest of the map. No random probes were used. To go from a low chance of completion to a 100% chance is admirable to say the least.

Even on the "hardest" map (HARD10), the ATS agent doesn't use any random probes, giving it a 100% chance of completing it every time.

3 Literature Review

Minesweeper isn't the exact same game as Daggers and Gold, but they are very similar. The main difference is in Minesweeper, you only have 1 life. You screw up, you die, simple as that. This adds a hefty weight to using a random probe, so you'd want to minimise the need for one as much as possible.

In the article I read, the engineer used many of the same techniques as my agents use, using a Single Point inspection method to ensure all obvious cells are flagged, and then resorting to a multi-cell comparison method when no more simple deductions can be made. This multi-cell method is similar to the satisfiability test my ATS agent runs.

4 Evaluation and Conclusion

In conclusion, I feel I have met all the requirements of the specification, building a system that can utilise multiple algorithms in order to solve even the most complicated of maps. Random Probing, Single Point Inspection and Satisfiability Tests have all been implemented.

The literal naming issue regarding LogicNG proved problematic, as there was no documentation relating to the issue, however, the fix was relatively simple, even if it feels the issue shouldn't be there in the first place.

5 Running

To run the agents, you must first select a map. This is done quite simply. To select the map difficulty, pass an argument containing either 1, 2 or 3. These represent easy, medium and hard respectively. The second argument is the map number for that particular difficulty.

To run the .jar files, use the command
java -jar Logic[n].jar [map difficulty] [map number]

6 Bibliography

https://luckytoilet.wordpress.com/2012/12/23/2125/