Logical Agents

Daggers and Gold Sweeper

Introduction

The assignment asks to implement and evaluate a number of AI Logical agents. They should be able to perceive the worlds they occupy and use logical reasoning to play the Daggers and Gold Sweeper game (D&G). I have completed all basic requirements, as well as a couple of extensions. The basics are: Logic1 which uses the Random Probing Strategy and the Single Point Strategy; Logic2 which also uses the Satisfiability Test Strategy. The more valuable extension is Logic3 which uses clues for Gold Mines to guide its reasoning. The other extension is refreshing the output instead of adding to it which makes tracking the agent's progress easier.

Running

The provided jar files can be run via the standard command "java -jar LogicX.jar" where X is 1, 2, or 3. No parameters need to be provided. The World to be solved is parsed from the standard input. Enter a string which corresponds to the name of a World. Any input is converted to capital letters before determining the corresponding World. After that, enter the seed for the agent's random generator. This allows repeating runs. A random seed is used if the input is invalid (including an empty line). After the agents completes the game, the program returns to waiting for input World.

Design

In designing and developing the program for this practical, I have taken an Object-Oriented approach. The game world and the agents are all objects. There are three classes that implement core functionality, which is used by the other classes.

The 'util' package contains a few classes with supporting functionality that I have written over the past couple of years. Some util classes have been written for previous practicals but none of them were created specifically for this practical.

The classes in the other three packages ('ui', 'agent', 'task') are written for this assignment. The classes inside Task define how the world is represented in the system. The Agent package contains an abstract agent class and three other classes that inherit its functionality. The UI package contains only one class – Loop, which defines the main loop through which the user interacts with the program

Task

This package contains three classes and an enum. The World enum is taken from the example on studres. I have only added two game worlds of my own (custom1 and custom2). They are used for the evaluation of my extension.

A Cell represents a single cell in the grid of a D&G. Its fields say whether it contains a dagger, gold, or a clue; whether it is covered, marked, adjacent to gold, and resolved; and its coordinates in the grid. Once a cell is created, its coordinates and contents do not change for the rest of the game. Therefore, I have made these globally accessible constants. This provides easy access without risking changing the fields outside the class. The constructor for this class can only be accessed within the classes. There are three globally accessible static methods which create cells. The methods create either a cell with a clue, with a dagger, or with gold, but never with more than one. The other four boolean fields are accessed via setters and getters. These fields, however, can only be changed once. Once a cell is uncovered, it can not be recovered. Once a cell is marked, it cannot be unmarked. Once a cell is determine to be next to gold, it cannot be reverted. Once all the adjacent cells are open, this cell is resolved, and it cannot be unresolved. Apart from the accessors and the constructors, there are two methods which are used to get a string representation of a cell. One for the basic displaying, and one for displaying in the extension.

The Game class represents the current D&G. It keeps a two-dimensional array of Cells which is initialised in the constructor. The constructor takes a two-dimensional array of Strings which is expected to be the map of a World. There are many methods here that provide a variety of functionalities. Determining whether a game is over happens after every move made by the agent. Retrieving the top left cell or a random cell is used for each move the agent makes. The method ‘applyUntilPositive’ accepts a method which gets invoked on each cell in the game until the method returns a positive value which is in turn returned by this method. This is used in the Single Point Strategy (SPS) where the agent iterates over all cells until it finds one for which SPS is applied successfully. Retrieving a specific cell is used in this class and its inheritors when accessing the neighbours of a cell. The method checks if the coordinates are within the grid before accessing the cell.
When printing the board on the screen, the number of printed lines is retrieved along with the string representation of the board. The first line of the string is made up of dashes ‘-’. After that, the lines are written in pairs. The first line of the pair is a row in the grid. The second line is again dashes ‘-’. The dashes are used to separate lines. Vertical lines ‘|’ are used to separate horizontally adjacent cells.
All other methods in this class are used to create a knowledge base for the agent playing the game. The knowledge base (KB) is a string of a logic formula. The formula is made up of two conjuncted parts. The first part says that in all covered cells, there are R remaining daggers, where R is the difference between the total number of daggers and the number of daggers found by the agent. The method ‘dimacs’ takes a number n and an array list of Cells. It iterates all possible combinations of of n cells within the array list. It constructs a logic sentence in Disjunctive Normal Form. That is, each clause represents a possible configuration, and at least one of them needs to be true. Each clause contains all variables, the clauses differ by which variables are expected to be true, and which – false. Therefore, no two clauses can be true at the same time. The second part is a

conjunction of logic formulas. Each of those represents the information obtained from an unresolved clue. The method ‘perCellDimacs’ takes a Cell and calculates how many more daggers need to be found. It then invokes ‘dimacs’ to create the logic sentence for this cell. The method ‘dimacsForAllCells’ calls ‘perCellDimacs’ for each cell and returns the conjunction of all the returned logic sentences.
The method ‘toDimacs’ is the one that constructs the KB of all of these. Most of these methods have error handling code in the form of try-catch blocks. When the board is very large, the logic formulas get very large. In some cases, these may cause an Out of Memory Error. Such errors get caught, logged to a log file, and the method returns with an indication of failure (usually by returning null).

Finally, the class GameGold is a child class of Game. It adds methods that support Strategies that search for Gold. The constructor of a GameGold calls its super constructor first and then iterates its cells, determining which ones are next to a cell with gold. The ‘toString’ method is largely similar to its parent’s ‘toString’. The only difference is that the board is wider, to make room for the letter ‘m’ to be displayed in cells that are next to a gold **m**ine. The other methods are used to construct a logical sentence about gold that the agent is searching for.
It is assumed that the agent does not know the total number of gold mines. The agent knows only whether a cell is adjacent to a gold mine. The agent does not know the exact number of gold mines around a cell. Therefore, the logical sentence for a cell next to a gold mine says that at least one of the cardinal neighbours must be true. Conversely, the logical sentence for a cell that is not adjacent to gold mines says that all of the cardinal neighbours are false. The total knowledge base about gold (goldKB) is the conjunction these clauses. Again, out of memory errors are handled in a similar way and the agent handles the case when the KB or the goldKB is null.

Agent

This package contains four classes. AAgent is an abstract class providing core functionality used by all agents. The other three classes are the agents for each of the three parts of the practical. The agent keeps track of its life, how many daggers it has found, a reference to the current Game, and stores a generator for random numbers. The game and the generator are instantiated in the agent’s constructor. The generator is given a seed. The generator is used by the Random Probing Strategy (RPS) when deciding on a cell to probe. The seed is provided by the user. Therefore, repeated runs of the agent with the same world and the same seed produce the same results.

The AAgent class has ten methods that check the game over conditions, and implement RPS and SPS. The methods ‘isDead’ and ‘hasWon’ are called after every move the agent makes to determine if the game is over. An appropriate message is displayed in both cases (see ui.Loop). As per the practical specification, the first move the agent makes is the top left corner of the grid (with coordinates 0,0). The method ‘makeMove’ is an abstract method and each agent defines it in accordance with the strategies it uses. The method ‘moveAndShowBoard’ is the one called in ui.Loop. It invokes ‘makeMove’ and then prints the current game board. The ‘probe’ method uncovers a cell that it is given, displays an appropriate message, and adjusts the life of the agent if needed.
The other methods implement RPS and SPS. RPS simply gets a random Cell from the Game (via

the agent's random generator), and probes it. SPS makes use of the Game's 'applyUntilPositive' method. It applies the Single Point Strategy for all cells, until one of them succeeds (or all fail). The method 'spsOn' attempts to perform SPS on a given cell. It fails if the cell is covered, resolved, or contains a dagger. Otherwise, it tries counts how many more daggers need to be found. If that number is zero, it probes all neighbours that are not marked. If that number is equal to the number of covered neighbours that are not marked, it marks them all. If no neighbours are marked or probed, then the method fails. This happens when the neighbours of the covered cell have been inspected from one of their other neighbours. For example, if we have a bunch of 0s as clues in the bottom right corner. This algorithm will take the left-most and upper-most one of them and probe the neighbours. On the agents next move, the adjacent zeros will not be marked as resolved but will have had their neighbours uncovered. If the method succeeds, then the agent would have made an empty move.
A lot of the methods here, and in the children classes return an integer that may seem meaningless at first glance. This number represents the number of lines this method has printed to the screen. Methods often add their number of lines to the ones returned by the methods they invoke. For example, RPS prints one line and then probes a cell, so it returns 1 + the value returned by probe. The method 'moveAndShowBoard' returns the total number of lines printed while the agent was making a move. This is used by ui.Loop for interactiveness (explained in the next section of this report).
The method 'printFail' is executed when an agent's current strategy fails. It simply display a failure message and returns the number of lines printed.

The Agent classes each have a one-line main method which starts the program and specifies that the agent that is to be used for playing the game. These classes also implement the 'makeMove' move method as a sequence of strategies that the agent attempts.

Agent1 is simple, it simple tries SPS. If it fails, it prints a failure and tries RPS (which always succeeds). It is worth noting that the 'makeMove' method accounts for the one line each strategy prints if it fails. That is the line with the name of the strategy. All agents account for this line printed by each strategy, if the strategy fails.

Agent2 is the class that contains the implementation of the Satisfiability Test Strategy (ATS). It makes use of the Game's 'toDimacs' method to get a string of the logical formula representing the knowledge base (KB). The methods in this class deal with transforming this formula into cnf, converting it to the format used by DIMACS, and then testing for satisfiability with different assumptions. The method 'clauses' takes the KB, converts it to CNF and invokes 'encodeInts' to create an array of integer arrays. Each of the integer arrays contains the literals of one clause from the CNF version of the KB. These integer arrays can now be used as clauses for the SAT4J solver. When encoding the formula as integers, the algorithm has to deal with the extra variables created by LogicNG during the transformation into CNF. In order to encode them as integers, I assign them values and store them in a hash map, so that repeated uses of the same variable are replaced by the same integer. The first number assigned is (NxM). The number then increases by one after each assignment. The cells in the game have ids from 0 to (NxM -1) where N and M are the dimensions of the grid. Thus NxM is the total number of cells in the grid.
The method 'notSatisfiable' takes the clauses and an integer to test. It creates a SAT4J solver, gives it the clauses and then a clause containing only the integer to test. This method returns true if the

solver does not find a solution.
The method ‘timedCall’ is used to impose a time limit on the CNF transformation and the satisfiability test. If a method takes too long to execute, it will be interrupted, and ATS will fail. The purpose of this is to speed up agent execution. In most cases, if it does not finish within a few seconds, it will fail.
The method ‘ATS’ performs the Satisfiability Test Strategy. It first gets the KB from the Game and transforms into clauses. It then gets a set of cells that represent the horizon. The horizon is the collection of covered cells that are adjacent to uncovered cells. These are the cells that the agent has information for. The algorithm iterates these cells, and uses ‘notSatisfiable’ on each to test if the cell should be marked as containing a dagger. Marked cells are removed from the horizon set and the remaining are tested if they should be probed. If a cell is probed, the method returns early. This way ATS probes at most one cell.

Lastly, Agent3 is the class that implements SPS and ATS to search for gold (goldSPS and goldATS) before searching from daggers. It extends Agent2, so it has access to the default ATS. The two gold algorithms are similar to SPS and ATS. The method ‘goldSPS’ iterates uncovered cells next to gold mines. If such a cell has only one covered cardinal neighbour, and such a cell has no uncovered cardinal neighbours with gold, then the covered cardinal neighbour must contain the gold.
The method ‘goldATS’ uses GameGold’s method to get the goldKB described at the end of the previous subsection in this report. The method then uses the inherited ‘clauses’ to transform the goldKB into an array of integer arrays. Finally, ‘goldATS’ iterates the cells in the horizon, assumes they are false, and test for satisfiability. In the case of contradiction, the assumption is wrong, so they must be true. Any gold cells found this way are probed.

UI

This package contains only one class – Loop. This class is designed to be instantiated and run once. It defines the main loop through which the user interacts with the logical agents. The constructor of Loop takes a class that extends AAgent or the AAgent itself. It then saves a reference to the constructor of that Agent. The method ‘iterate’ is the only method visible to other classes. It contains the loop which wait for user input. If the input is ‘exit’, the loop breaks. Since all three main methods only execute the ui.Loop.iterate method once, breaking the loop will also exit the program. Any other input is parsed as the name of a World from the World enum. If parsing fails, an appropriate message is displayed and the loop goes back to waiting for input. Before displaying the error message however, the last three lines of the terminal are deleted. These are the input line, the line prompting the user for input, and the line where any previous error message would be. After deletion, printing the error message on one line, and the prompt on a second, puts the cursor on the third line where it used to be. So, from the user’s perspective, the program has consumed the input and has shown an error. The next input happens on the same line as the previous input.
Deleting the last line is done by moving the cursor up, printing blank characters (spaces) to cover the previous output, and then positioning the cursor back at the start of the line. Repeating this in a loop, causes multiple previous lines to be cleared.
When the user’s input is successfully parsed as a World enum, the method ‘nextIteration’ is called. That method first calls ‘instantiateAgent’ to create a new instance of AAgent. That method, in turn, prompts the user to enter a seed for the agent’s random generator. If the input is not a valid long

number, a random such is created. The previously saved constructor is used to create an AAgent instance. ‘nextIteration’ then continues by making the first move (probing the top-left corner). After that, it enters the game loop. In the loop, a check is made to see if the game is over. If it is not, then the program waits for the user to input a character. If it is a capital Q, the game loop will quit, and the program will return to waiting for input world. If the input character is instead a new line (returned when ‘Enter’ is pressed), the program will clear the last few lines and invokes the agent’s ‘moveAndShowBoard’ method. That method will display the agent’s decision process, its move, and the new board. The method will return the number of lines it has printed. That number will then be used in determining how many lines need to be cleared, after the user presses enter again. This way, the logical agent’s process may be observed without filling the terminal with boards.

Note: when storing, Cells are indexed first by row i, then by column j. In the lectures however, they are index first by column x, then by row y. Thus, When displaying cell coordinates, they appear column first, row second. For example, cell with i=3, j=8, and clue=2 will appear as “8 3 clue: 2”.

Examples & Testing

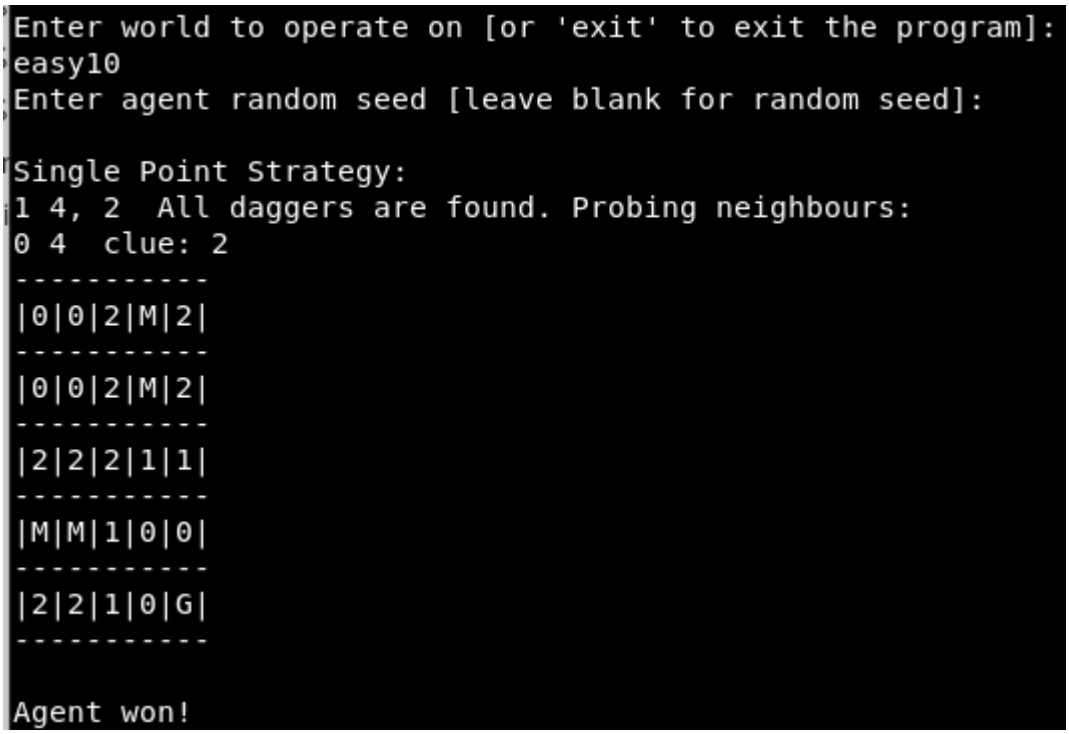
I tested the system by running the agents on the provided Worlds and on two Worlds that I created. I have not included screenshots of all runs, as they are not vastly different. All screenshots are in this and in the next section.

```
> java -jar Logic1.jar

Enter world to operate on [or 'exit' to exit the program]:
easy1
Enter agent random seed [leave blank for random seed]:
0
Single Point Strategy:
4 2, 2  All daggers are found. Probing neighbours:
4 1  clue: 2
-----------
|0|0|0|1|C|
-----------
|0|G|1|2|2|
-----------
|0|1|2|M|2|
-----------
|0|1|M|3|M|
-----------
|0|1|1|2|1|
-----------

Agent won!
```

Agent1 has solved the D&G. EASY1, seed 0.

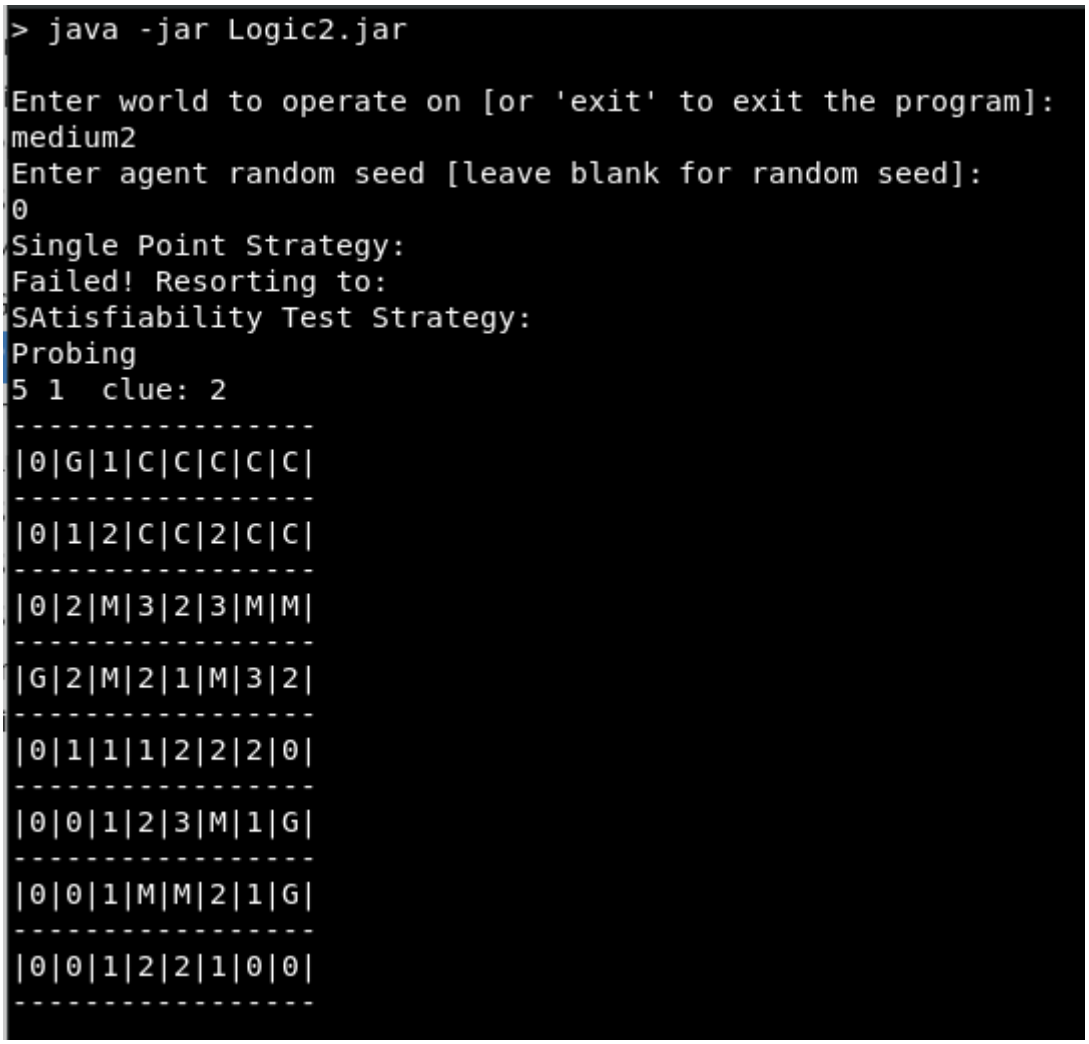


In the same execution as above, Agent1 has solved D&G EASY10 with a random seed.

```
Enter world to operate on [or 'exit' to exit the program]:
hard4
Enter agent random seed [leave blank for random seed]:
0
Single Point Strategy:
7 10, 3  All daggers are found. Probing neighbours:
8 11  clue: 1
-------------------------
|0|0|0|0|0|2|M|2|0|0|0|G|
-------------------------
|0|0|0|1|1|3|M|2|0|0|0|0|
-------------------------
|1|2|1|2|M|2|1|1|1|1|1|0|
-------------------------
|M|3|M|2|1|2|1|1|1|M|1|0|
-------------------------
|M|3|1|1|G|1|M|3|3|2|1|0|
-------------------------
|1|1|1|1|1|1|2|M|M|2|1|0|
-------------------------
|0|0|2|M|2|0|1|3|4|M|2|1|
-------------------------
|0|1|3|M|2|0|0|1|M|2|2|M|
-------------------------
|1|2|M|2|1|0|1|3|3|3|2|2|
-------------------------
|2|M|4|2|1|0|1|M|M|3|M|1|
-------------------------
|3|M|4|M|2|0|1|3|M|3|1|1|
-------------------------
|M|2|3|M|2|0|0|1|1|1|0|G|
-------------------------

Agent won!
```

Agent1 solved HARD4 with seed 0.



Agent2 applies ATS after SPS fails and sees that it can probe the cell in column 5, row 1.

```
Enter world to operate on [or 'exit' to exit the program]:
hard10
Enter agent random seed [leave blank for random seed]:
1
Single Point Strategy:
10 10, 0  All daggers are found. Probing neighbours:
11 11  clue: 0
-------------------------
|0|2|M|4|M|2|M|1|G|1|M|1|
-------------------------
|0|3|M|M|2|3|3|3|1|1|2|2|
-------------------------
|1|3|M|3|1|1|M|M|1|0|1|M|
-------------------------
|M|2|1|1|0|1|2|2|2|1|2|1|
-------------------------
|3|3|1|0|0|0|1|1|2|M|1|0|
-------------------------
|M|M|2|1|1|1|1|M|2|1|2|1|
-------------------------
|3|M|2|1|M|2|2|2|1|G|1|M|
-------------------------
|1|1|1|2|2|3|M|1|0|0|1|1|
-------------------------
|0|0|0|1|M|2|1|2|1|1|0|0|
-------------------------
|0|G|0|1|1|1|1|3|M|2|0|0|
-------------------------
|0|1|1|1|1|1|2|M|M|2|0|0|
-------------------------
|0|1|M|1|1|M|2|2|2|1|0|0|
-------------------------

Agent won!
```

Agent2 solves HARD10 with seed 0.

```
> java -jar Logic3.jar

Enter world to operate on [or 'exit' to exit the program]:
hard10
Enter agent random seed [leave blank for random seed]:
1
Single Point Strategy for gold.
Found gold
8 0  contained gold. +1 Life
-------------------------------------------------
| 0 | 2 | M | 4 | M | 2 | M | 1m| G | C | C | C |
-------------------------------------------------
| 0 | 3 | M | M | 2 | 3 | 3 | 3 | C | C | C | C |
-------------------------------------------------
| 1 | 3 | M | 3 | 1 | 1 | M | M | C | C | C | C |
-------------------------------------------------
| M | 2 | 1 | 1 | 0 | 1 | 2 | 2 | C | C | C | C |
-------------------------------------------------
| 3 | 3 | 1 | 0 | 0 | 0 | 1 | 1 | C | C | C | C |
-------------------------------------------------
| M | M | 2 | 1 | 1 | 1 | 1 | M | 2 | 1m| 2 | C |
-------------------------------------------------
| C | C | C | C | C | C | C | 2 | 1m| G | 1m| C |
-------------------------------------------------
| C | C | C | C | C | C | C | 1 | 0 | 0m| 1 | C |
-------------------------------------------------
| C | C | C | C | C | C | C | 2 | 1 | 1 | 0 | C |
-------------------------------------------------
| C | C | C | C | 1 | C | C | C | C | C | C | C |
-------------------------------------------------
| C | C | C | C | C | C | C | C | C | C | C | C |
-------------------------------------------------
| C | C | C | C | C | C | C | C | C | C | C | C |
-------------------------------------------------
```

Agent3 uses SPS for gold to find a gold mine.

```
Enter world to operate on [or 'exit' to exit the program]:
custom2
Enter agent random seed [leave blank for random seed]:
0
Single Point Strategy for gold.
Failed! Resorting to:
Gold ATS:
3 3  contained gold. +1 Life
---------------------------------
| Gm| Gm| 1m| C | C | C | C | C |
---------------------------------
| 1m| 1m| 2 | C | 3 | 3 | C | C |
---------------------------------
| C | C | 3 | C | C | C | C | C |
---------------------------------
| 3 | M | C | G | 3m| C | 3 | 0 |
---------------------------------
| 2 | M | C | C | 3 | M | 2 | 0 |
---------------------------------
| 1 | 2 | 3 | C | 2 | 2 | 2 | 1 |
---------------------------------
| 0 | 1 | C | C | 2 | 2 | M | 1 |
---------------------------------
| 0 | 1 | C | C | D | 2 | C | 1 |
---------------------------------
```

Agent3 uses gold ATS to find the only possible location for a gold mine.

Evaluation

Overall, my implementation achieves all the required basic parts, as well as two extensions. The screen clearing before the next output supports following the logical agent's progress through the game. The Gold Mine searching furthers the analysis although the result of that is unexpected.

There are several improvements that I wish I had the opportunity to do. One of them is an 'undo'. The user would be able to enter a command (or even simply the 'U' character) and cause the system to undo the last move made by the agent. This would have to take into account random moves and make sure that if a random move is undone, the same move is redone upon continuation. Another improvement would be allowing the user to give a move for the agent. The agent will make that move, instead of deciding on its own. These two would allow analysing odd or edge cases. Keeping a count of the number of random moves the agent makes would also have been a useful improvement. In this evaluation, I kept this count manually.

There are a couple of more improvements that are described in the following subsections. These subsections evaluate and make conclusions about the behaviour of the agents.

Agent1

This agent uses SPS and RPS. SPS is a powerful strategy on its own. It is fast and resolves a lot cells. Its effectiveness drops as the number of daggers increases. Even so, probing a random cell

often gives SPS a new resource to explore. With a little luck, even this agent can solve the hardest of D&Gs.

```
> java -jar Logic1.jar

Enter world to operate on [or 'exit' to exit the program]:
hard10
Enter agent random seed [leave blank for random seed]:
4
Single Point Strategy:
10 10, 0  All daggers are found. Probing neighbours:
11 11  clue: 0
-------------------------
|0|2|M|4|M|2|M|1|G|1|M|1|
-------------------------
|0|3|M|M|2|3|3|3|1|1|2|2|
-------------------------
|1|3|M|3|1|1|M|M|1|0|1|M|
-------------------------
|M|2|1|1|0|1|2|2|2|1|2|1|
-------------------------
|3|3|1|0|0|0|1|1|2|M|1|0|
-------------------------
|M|M|2|1|1|1|1|M|2|1|2|1|
-------------------------
|3|M|2|1|M|2|2|2|1|G|1|M|
-------------------------
|1|1|1|2|2|3|M|1|0|0|1|1|
-------------------------
|0|0|0|1|M|2|1|2|1|1|0|0|
-------------------------
|0|G|0|1|1|1|1|3|M|2|0|0|
-------------------------
|0|1|1|1|1|1|2|D|M|2|0|0|
-------------------------
|0|1|M|1|1|M|2|2|2|1|0|0|
-------------------------

Agent won!
```

Agent1 has solved HARD10 with a seed of 4, and it did only 8 random probes (manually counted). So, in a 12 by 12 grid, the agent randomly probed 8 out of 144 cells. After running it several times with a random seed, the average number of random probes was 5.

Agent2

This agent uses ATS if SPS fails. ATS is a more complex strategy and is, unfortunately, not as good as SPS. It takes longer to run and usually gives information about 1 to 3 cells. Compare that to SPS which can often mark or probe as much as 4 cells.

```
Enter world to operate on [or 'exit' to exit the program]:
medium10
Enter agent random seed [leave blank for random seed]:
0
Single Point Strategy:
1 5, 2  All daggers are found. Probing neighbours:
0 5  clue: 2
0 6  clue: 2
1 6  clue: 2
2 6  clue: 2
-----------------
|0|0|1|M|M|1|0|G|
-----------------
|1|1|1|2|2|2|1|1|
-----------------
|M|1|G|0|0|1|M|1|
-----------------
|3|3|1|0|G|1|1|1|
-----------------
|M|M|2|2|2|1|0|0|
-----------------
|2|2|2|M|M|1|0|0|
-----------------
|2|2|2|C|C|C|C|C|
-----------------
|C|C|C|C|C|C|C|C|
-----------------
```

Even so, ATS is better than guessing at random. With it, the agent is guaranteed to probe a cell without a dagger. In solving HARD10 with a seed of 4, this agent did 6 random moves. After running it several times with a random seed, the average number of random probes was 3.

A drawback in my implementation is that if the knowledge base is too large, ATS fails by default. A good improvement would be storing a partial KB before trying adding the next clause. If the KB is stored in a file, only parts of it could be accessed at any one time. Thus, it would avoid running out of memory. ATS could be performed on partial KB or the algorithm could be extended to access a file for the parts of the logic formula. Either way, ATS would give more results (because it is using more clauses) at the cost of speed, because file access and cutting logical formulas will require a lot of computation.

Agent3

This agent uses goldSPS and goldATS before using the normal SPS and ATS. This is counter intuitive, as the gold searching algorithms should support the normal searching, not the other way around. However, I have found that searching for gold mines is not as effective as straight up searching for daggers. There are a couple of reasons behind this. First, the agent does not know the exact number of gold mines around a cell. Take for example a cell A with clue 2 and a cell B between two gold mines. Some of the neighbours of A are already probed and a dagger is found. Some of the neighbours of B are also probed and a gold mine is found. There are still one dagger

and one gold mine. However, the agent is only aware of the one dagger, so the regular SPS can help while the gold SPS will not yield results. Furthermore, taking into account the fact that there are no daggers next to gold mines, cells that are adjacent to a gold mine will often have small clues (typically in the range of 1 to 3). Such cells can be quickly resolved by regular SPS and thus, the gold will be found as a neighbour that can be safely probed. I initially configured Agent3 to use the regular algorithms before the gold seeking ones but I could hardly verify them. Using gold seeking before regular logic, gives some results but it is likely that SPS and ATS would have found those as well.

Given all this, I have an idea of how it could be improved. I have the extension, as it was described in the practical specification. However, if instead of looking at cardinal neighbours only, the algorithm looked at all neighbours, then it would have much more information to work with. Further, if the total number of gold mines in the game is known, and each cell says how many gold mines are near it, then the gold seeking algorithms would be just as effective as the regular ones because they have the same amount of information.

The screenshots previous section demonstrate that gold SPS and goldATS both work. Both custom maps were created specifically for the purpose of testing these algorithms. There may be combinations of existing worlds and specific values for the seeds which also let the gold seeking algorithms do some work (hard10 and a seed 1 for example).

Literature Review

Having played a lot of minesweeper games, I know SPS is a strategy used all the time. I only did partial ATS just by narrowing down which cells cannot have mines in them. I looked into several sources and most algorithms can be generalized into ATS.

This short study[1] looks into how an AI can help a human player solve a minesweeper. It never uses the terms SPS or ATS but these are the algorithms described.

This paper[2] describes the technical details of SPS and ATS. It defines Naive Single Point and Double Set Single Point strategies that differ from my implementation. However, that is only because their algorithms iterate the cells in a different manner. In practice, my implementation achieves the same result. Their ATS algorithms do have an advantage over mine. The main improvement is partitioning the set of cells under inspection. It is a step up from my suggestion in the evaluation section and it reminds me of a divide and conquer approach.

Word count without Bibliography: 4476.

Bibliography

[1] Solving Minesweeper
Magnus Hoff
https://magnushoff.com/minesweeper/

[2] Algorithmic Approaches to Playing Minesweeper
David J. Becerra
http://nrs.harvard.edu/urn-3:HUL.InstRepos:14398552