CS5011-Practical 3-Logic

170024030

November 21, 2017

1 List of implemented parts and extensions

Part I. Logic1.jar

Random Guessing Strategy (RGS)

Single Point Strategy (SPS)

Extension of Part I. Logic1_1.jar

Smart Random Guess Strategy (SRGS)

Part II. Logic2.jar

The Easy Equation Strategy (EES)

Part III. Logic3.jar

The Davis-Putnam-Logemann-Loveland Strategy (DLS)

2 Literature Review

Throughout the relatively short history of Artificial Intelligence, it has been significantly promoted and influenced by logical ideas[2]. Russel and Norvig [1] concluded that logical agents are designed to form a general class of representations of a complex world and use this information to suit myriad purposes. It is worth noticing that the goal for logical agents must be explicitly described. Agents with well-designed logic are also able to consecutively learn new knowledge about the environment and adapt to changes[1]. In this project, the agent update its knowledge every time a progress is made and deduce the next movement from the knowledge base. With the knowledge base, the logic agent can decide what to do by choosing the right move entailed by the knowledge base which is implemented by the third part of this project, the propositional logic method.

3 Part I.

3.1 Design

3.1.1 The PEAS agent

The environment, the knowledge base, and the causal reasoning can be characterized by these four aspects:

- Performance measure: -1 nettle when the agent marks a cell, -1 unknown cell and +1 known cell when the agent probe or mark a cell, the agent dies when a nettle cell is uncovered. The game ends either when the agent dies or when the agent managed to mark all nettles.
- Environment: A $m \times n$ grid of cells. The size of the map is determined by the argument given by the user. The agent always starts in the cell labeled $[0, 0]$. The locations of the nettles are set by the given maps. For the given maps, easy-level worlds are 5×5 with 5 nettles; medium-level worlds are 9×9 with 10 nettles; hard-level worlds are 10×10 with 20 nettles.
- Actuators: The agent can probe a cell or mark a cell. The agent will instantly die if it probes a cell that contains nettle. The agent cannot mark or probe a cell that is already marked or uncovered.
- Sensors: The agent only has one sensor to find out the number of nettles in a cell's eight neighbors.

3.1.2 Object design

In the first part, an object called Agent_RGS_SPS is designed as a basic knowledge-based agent to handle the nettle world. Its instance variables include the answer map, the covered map, total number of unmarked nettle, the x and y scale of the map, three array list to store collections of known and unknown frontiers and all unknown cells. The uncovered cells will be shown as the numbers they contain; the covered cells are "?"; the nettle flag is "N"; uncovered nettle is "*".

```
0 0 0 2 ?
0 0 0 2 ?
1 2 1 2 ?
? ? ? ? ?
? ? ? ? ?
```

Figure 1: A state of game in the 5×5 nettle world. The known frontiers are marked as red, and the unknown frontiers are marked as green.

The answer map is used to probe cells and get the information contained in that cell; the covered map is the knowledge base that stores all the uncovered and marked cells.

The known frontiers are the uncovered cells that have one or more neighbors; the unknown frontiers are the covered cells (not marked) that are adjacent to covered cells. The frontier array lists are set to optimize the SPS and other strategies.

3.1.3 Implementation of RGS

The implementation of RGS is quite straightforward. To begin with, all the covered cells are stored in a dynamic list. The agent then generates a random integer within the length of the unknown list, and the probe method is then invoked with that random cell. In this project, since RGS is the only method that will probably uncover cells that contains nettle, it is necessary to check if the agent is still safe after every time the RGS is resorted to. Therefore, if the agent uncovers a nettle cell, the boolean attribute of the agent object will be set to false from its default setting, true. The main method that instantiates the nettle world agents will check if the agent is safe every time it makes a progress.

3.1.4 Implementation of SPS

For SPS, instead of going through all covered cells, I make the agent focus on unknown frontiers only, because the SPS strategy only works on them. For each loop, the working sequence is listed as followed.

1. Scan all unknown frontiers one by one.
2. For each unknown frontier, check its known neighbors.
3. If the cell has all free neighbors: probe this cell; if the cell has all marked neighbors: mark this cell.
4. Repeat until no other change can be made.
5. Then resort to RGS.

Whenever a progress is made by SPS or RGS, a statement will be printed indicating the action of the agent followed by the current state of the game. To make step 4 and step 5 work, the SPS method will always return a sentinel boolean value: true if any change is made and false if failed to probe or mark any cell. These sentinel value are then fed to the loop in the game infrastructure. If the SPS does not work for the current state, the agent will then resort to RGS, and the outer loop will bring the additional information brought by RGS to SPS again until all nettles are marked or a nettle cell is uncovered. If a nettle cell is marked, the main-body loop will also stop.

3.1.5 Implementation of SRGS

In addition to RGS, the probability is considered for the covered cells, and this method will choose the safest cell from the covered frontiers to probe. The agent class SRGS is designed with this strategy. The probability of a covered frontier being nettle-free $p_{[x,y]}$ can be calculated from its uncovered safe neighbors, given by

$$p_{[x,y]} = 1 - \frac{v_{[x,y]} - m_{[x,y]}}{t_{[x,y]}} \tag{1}$$

where $[x, y]$ is the location of the cell's uncovered safe neighbor, v is the number of nettle contained in the cell's neighbors, m is the number of nettles already marked in the cell's neighbors, and t is the total number of the cell's neighbors.

The work sequence for this random strategy is listed below:

1. Scan all unknown frontiers one by one.
2. For each unknown frontier $[m, n]$, scan its uncovered safe neighbors $[x, y]$.
3. For each of its uncovered safe neighbors, calculate $P_{[x,y]}$.
4. $P_{safe[m,n]}$ is the maximum value of all $p_{[x,y]}$around $[m, n]$.
5. Store all unknown frontiers $[m, n]$ in a list according to $P_{safe[m,n]}$ with a decreasing order.
6. Randomly probe an unknown frontier from the first $\frac{1}{3}$of the ranked list.

The reason why the agent randomly picks first $\frac{1}{3}$ of the ranked list rather than probe the first cell is that, if the first cell contains nettle, the agent will get stuck and never solve the problem. In this way, it is more likely to probe a safe cell than the totally random guess by adding knowledge-based uncertainty measure to the unknown frontiers.

3.2 Examples and Testing

To test and compare the performances of SPS_RGS agent and SPS_SRGS agent, "Logic1.jar" and "Logic1_1.jar" are applied with all the easy and medium maps for 10 times respectively. The success rates are shown in table 1, and the average number of random guesses needed to win the game are shown in table 2.

For easy map 1, 4 and medium map 1, 2, 4, the agent does not need random guess to win the game, but for the rest of the maps, with SPS along cannot solve all the states. The average success rate of SPS_SRGS agent is higher than SPS_RGS agent.

Table 1: The success rate of SPS_RGS agent and SPS_SRGS agent applying with given maps. The rate is calculated by running the program for 10 times with each map.

	Easy					Medium				
	1	2	3	4	5	1	2	3	4	5
SPS+RGS	100%	10%	50%	100%	70%	100%	100%	30%	100%	80%
SPS+SRGS	100%	50%	100%	100%	80%	100%	100%	20%	100%	70%

Table 2: The average success rate of SPS_RGS agent and SPS_SRGS agent applying with given maps. The average value is calculated of 10 success cases with each agent and each map.

	Easy					Medium				
	1	2	3	4	5	1	2	3	4	5
SPS+RGS	0	1.2	1	0	1.1	0	0	5.4	0	4.6
SPS+SRGS	0	2.5	1	0	1.2	0	0	4	0	5

3.3 Evaluation

As is shown in the tables above, the winning rate gets smaller when more random guesses are required to solve the problem. Although SRGS is designed as a knowledge-based enhanced version of RGS, for some of the problems, it does not always show better performance, SRGS only make random guess inside the covered frontier cells. In some state of the game, the least dangerous covered cell in the frontier is still more dangerous than average non-frontier cells.

An optimized algorithm should also take the total amount of unmarked nettles into consideration and calculate the probability for non-frontier cells as well.

3.4 Running

This part includes two jar files.

To apply RGS_SPS agent with given maps, use

"java -jar Logic1.jar <level> <map_number>".

To apply SRGS_SPS agent with given maps, use

"java -jar Logic1_.jar <level> <map_number>".

The valid level arguments are "easy", "medium", "hard" and "master".

For "easy", "medium" and "hard" level, the valid arguments are from 1 to 5; there is only one map in "master" level.

4 Part II.

4.1 Design

4.1.1 Object design

The agent object for this part, Agent_RGS_SPS_EES, is a subclass of Agent_RGS_SPS in the first part. In addition to its superclass, four methods are implemented for the EES, which are discussed as followed.

4.1.2 Implementation of EES

The work sequence for each loop of the knowledge-based agent in this part is listed below:

1. Scan all unknown frontiers one by one.
2. For each unknown frontiers, check its known neighbors.
3. If the cell has all free neighbors: probe this cell; if the cell has all marked neighbors: mark this cell.
4. Repeat until no other change can be made.
5. Gather all pairs of bordering cells in the known frontiers and apply EES.
6. If no changes can be made, resort to SRGS.

The first problem is to find the pairs of bordering cells. By definition, the bordering cells are a pair of cells in which case one's adjacent covered cells are contained by the other's neighbors. To find all the pairs, I used a two-layered for loop to scan all pairs and invoke the "isContained" method to find out if these two frontiers are bordering pairs. This method returns three possible outputs: 0 if no contained; 1, if contained and the first one has more neighbors; 2, if contained and the second one has more neighbors. For each pair of bordering cells, the agent computes the total difference in undiscovered nettle value, given by

$$diff = |(v_{[x,y]} - m_{[x,y]})(v_{[k,j]} - m_{[k,j]})| \tag{2}$$

where $[x, y]$ and $[k, j]$ are the locations of the bordering pair, v is the number of nettles contained in the cell's neighbors, m is the number of nettles already marked in the cell's neighbors.

If $diff = 0$, "probeEes" method is invoked to uncover all the cells adjacent to only one of the two; if $diff = v_{[x,y]} - v_{[k,j]}$ "markEes" method is then invoked to mark all the cells only adjacent to one of the two cells; otherwise, nothing happens.

The game infrastructure remains largely the same as Part I. The EES method will also return a sentinel boolean value to indicate whether any progress is made by EES for the current state. If Both SPS and EES do not work for the current state, let the agent do a random guess. If the agent survives the random guess, grab the additional information from random guess and go through the work sequence again.

4.2 Examples and Testing

For all the given maps, the RGS_SPS_EES can pass without random guess. All the occurred states can be perfectly solved with the combination of SPS and EES. To test the performance of this agent, a more complex game is introduced in the master level (Fig. 2). The running time and numbers of different strategies applied are listed in table 3. The success rate for this map is 15% for 20 tests.

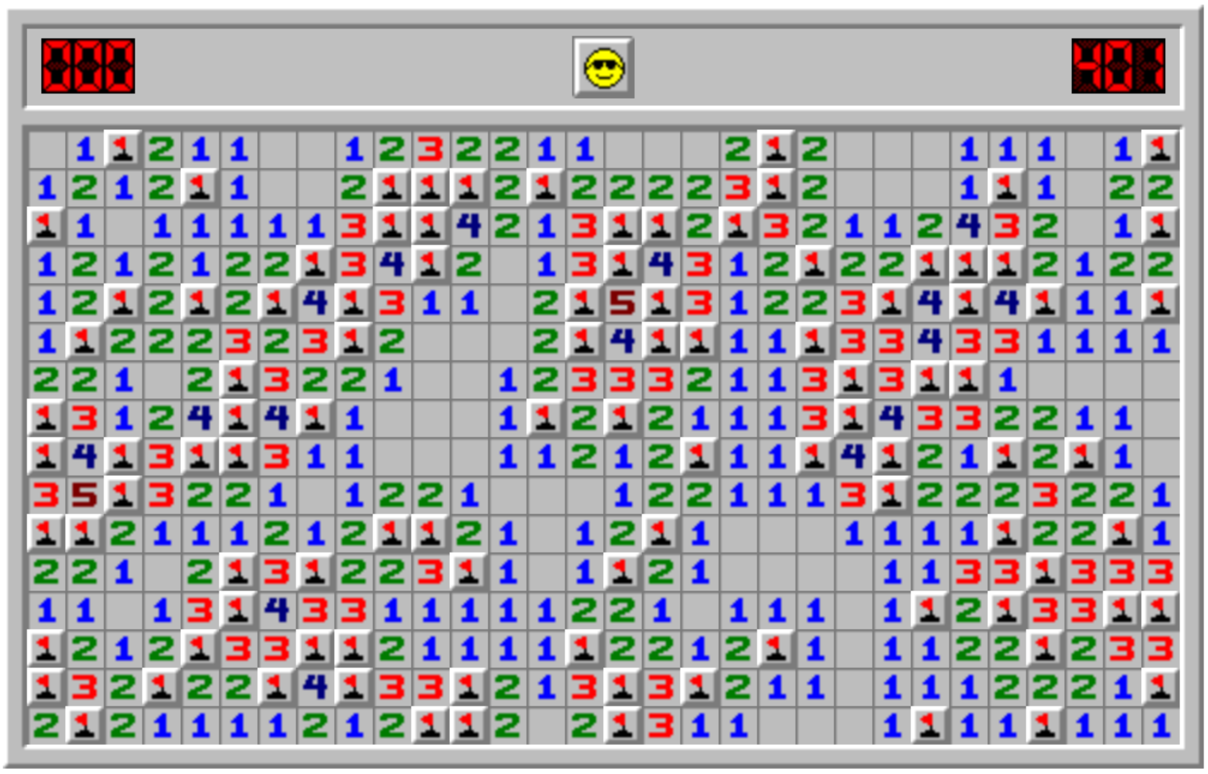


Figure 2: A real game that I played. This game map is stored in master level to test the performance of EES and DLS.

Table 3: The statistics of running RGS_SPS_EES agent on the master map.

Test	Computing Time(ms)	n(RGS)	n(SPS)	n(EES)
1	481	5	273	30
2	495	3	305	8
3	522	2	306	8
4	496	4	304	8
5	505	7	299	8
Average	499.8	4.2	297.4	12.4

4.3 Evaluation

Generally speaking, the RGS_SPS_EES agent is sufficient for the given maps. However, when dealing with a more complex situation, it shows inability. However, this master map is not an ideal map to test the success rate of these agents because the agents in this assignment always start from the upper right corner. Therefore, the agent can only guess after it makes the first move. Further study can include more equally complex maps, but with a friendlier upper right corner.

Compared with the results in Part 1, the agent with additional EES implemented is enabled to handle more complicated maps and is less dependent on the RGS. The success rate is dramatically improved to 100% for all the given maps. It also shows potential to deal with much more complex nettle world (like the “master” map), although the success rate is relatively low, because the map is very "unfriendly" for agents started the upper left corner.

The qualitative analysis of its performance will be presented in the next section against the performance of RGS_SPS_DLS agent.

4.4 Running

This part includes one jar file.

To apply RGS_SPS_EES agent with given maps, use

"java -jar Logic2_.jar <level> <map_number>".

The valid arguments are the same as Part 1.

5 Part III.

5.1 Design

5.1.1 Object design

The agent object for this part, Agent_RGS_SPS_DLS, also extends Agent_SRGS_SPS in the first part. In addition to its superclass, four methods are implemented for the EES, which are discussed as followed.

5.1.2 Implementation of DLS

The work sequence for each loop of the knowledge-based agent in this part is listed below:

1. Scan all unknown frontiers one by one.
2. For each unknown frontiers, check its known neighbors.
3. If the cell has all free neighbors: probe this cell; if the cell has all marked neighbors: mark this cell.
4. Repeat until no other change can be made.
5. Generate the total proposition (KBU) with all the current known frontiers.
6. Scan all unknown frontiers.
7. For each unknown frontiers, apply DLS KBU and its single proposition.
8. If no changes can be made, resort to RGS.

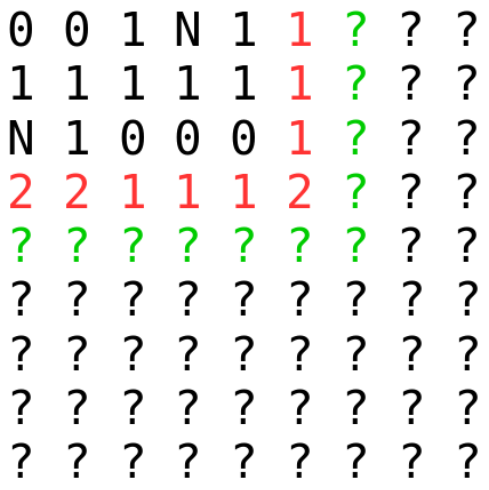
The total proposition is a complex logical proposition that contains all the knowledge from uncovered cells from the current state. The complete backtracking algorithms are given by the AIMA's implementation of the DPLL, which basically test if the first complex proposition entails the other one, proved by contradiction.

The major part of this implementations is designed to generate the total logical proposition according to the syntax given by AIMA's code. To get this done, "getSingleKBU" method is used to generate the KBU sentence for a single known frontier, and the "getKBU" method combine them and create the total KBU. A tricky part of "getSingleKBU" is to find all the possible setups around a known cell, given the number of unmarked nettles. I used a binary coding strategy to do this. For example, if the known cell has the information as three and has 8 unmarked neighbors, I generate a collection of 8-digit binary numbers, from 00000000, 00000001, 00000010 to ... to 11111110 and 11111111. Get rid of all the numbers with wrong numbers of '1', to give the right numbers of remaining nettles. This collection of binary numbers can, therefore, represents different setups around a known frontier. And then it is easy to transfer these binary combinations to logical propositions. Then both the positive and negative single propositions for each unknown frontier are checked if they satisfy the global KBU. If the positive single proposition does not satisfy, mark the cell; if the negative single proposition does not, uncover the cell; if both positive and negative proposition satisfies the KBU, go to the next unknown frontier cell.

Also, the game infrastructure is similar to Part II. The DLS method returns a sentinel boolean value as well to indicate whether if the agent should resort to SRGS.

5.2 Examples and Testing

The RGS_SPS_DLS agent works well with all the given map except for Medium3, where the agent will have to deal with a ten-possibility proposition and the lab computer got stuck and run out of memory.



```
0 0 1 N 1 1 ? ? ?
1 1 1 1 1 1 ? ? ?
N 1 0 0 0 1 ? ? ?
2 2 1 1 1 2 ? ? ?
? ? ? ? ? ? ? ? ?
? ? ? ? ? ? ? ? ?
? ? ? ? ? ? ? ? ?
? ? ? ? ? ? ? ? ?
? ? ? ? ? ? ? ? ?
```

Figure 3: A state of game when the RGS_SPS_DLS agent gets stuck on the lab computer.

For the state shown in fig. 3, the KBU is

KBU =((N_0_6&~N_1_6) | (~N_0_6& N_1_6))

& ((N_0_6&~N_1_6&~N_2_6) | (~N_0_6& N_1_6&~N_2_6) | (~N_0_6&~N_1_6& N_2_6))

& ((N_1_6&~N_2_6&~N_3_6) | (~N_1_6& N_2_6&~N_3_6) | (~N_1_6&~N_2_6& N_3_6))

& ((N_4_0&~N_4_1) | (~N_4_0& N_4_1))

& ((N_4_0&~N_4_1&~N_4_2) | (~N_4_0& N_4_1&~N_4_2) | (~N_4_0&~N_4_1& N_4_2))

& ((N_4_1&~N_4_2&~N_4_3) | (~N_4_1& N_4_2&~N_4_3) | (~N_4_1&~N_4_2& N_4_3))

& ((N_4_2&~N_4_3&~N_4_4) | (~N_4_2& N_4_3&~N_4_4) | (~N_4_2&~N_4_3& N_4_4))

& ((N_4_3&~N_4_4&~N_4_5) | (~N_4_3& N_4_4&~N_4_5) | (~N_4_3&~N_4_4& N_4_5))

& ((N_2_6& N_3_6&~N_4_4&~N_4_5&~N_4_6) | (N_2_6&~N_3_6& N_4_4&~N_4_5&~N_4_6)
| (~N_2_6& N_3_6& N_4_4&~N_4_5&~N_4_6) | (N_2_6&~N_3_6&~N_4_4& N_4_5&~N_4_6)
| (~N_2_6& N_3_6&~N_4_4& N_4_5&~N_4_6) | (~N_2_6&~N_3_6& N_4_4& N_4_5&~N_4_6)
| (N_2_6&~N_3_6&~N_4_4&~N_4_5& N_4_6) | (~N_2_6& N_3_6&~N_4_4&~N_4_5& N_4_6)
| (~N_2_6&~N_3_6& N_4_4&~N_4_5& N_4_6) | (~N_2_6&~N_3_6&~N_4_4& N_4_5& N_4_6))

The last term alone contains 50 single logic propositions because there are three nettles in five cells, and the total possibility is $C_3^5 = 10$ which takes too much memory to compute and the lab computer was not able to handle. The similar case also happens when applying to the "master" map. A file named "test.java" is included in the source file to double-check the reason for this problem.

5.3 Evaluation

Generally speaking, RGS_SPS_DLS agents is capable of dealing with most of the given maps without guessing, except for "medium3". Compared with the agent in Part 1, implementation of DLS makes the agent much more reliable. As is stated in the last section, the space efficiency of DPL is so poor that it failed to handle the "2-nettle-in-5-cell" situation. To quantitatively compare the performances of RGS_SPS_EES agent and RGS_SPS_DLS agent, both agents are tested with map "medium5" and "hard1", and the performance measures are listed in table 4. These two maps are selected because they require more non-SPS steps than the other given maps. As is shown in the table, EES illustrates significantly better time efficiency than DLS. And the number of applied EES or DLS steps indicates the decision made by EES is more optimal than DLS since it gives more information for SPS to work with.

Due to the problem with memory control of DLS, we cannot compare their performance with the "master" map. However, according to the performance measure from the given maps, it is safe to conclude that EES shows better performance in terms of time efficiency, space efficiency, and optimality.

Table 4: The performances of RGS_SPS_EES agent and RGS_SPS_DLS agent measured from two map.

Map	Agent	Computing Time(ms)	n(RGS)	n(SPS)	n(EES or DLS)
Medium 5	RGS+SPS+EES	29.8	0	20	8
	RGS+SPS+DLS	627.8	0	19	10
Hard 1	RGS+SPS+EES	50.6	0	43	2
	RGS+SPS+DLS	226	0	40	5

5.4 Running

This part includes one jar file.

To apply RGS_SPS_DLS agent with given maps, use

"java -jar Logic3_.jar <level> <map_number>".

The valid arguments are the same as Part 1.

References

[1] S. J. Russell and P. Norvig, "Artificial intelligence (a modern approach)," 2010.

[2] R. Thomason, "Logic and artificial intelligence," in *The Stanford Encyclopedia of Philosophy*, winter 2016 ed., E. N. Zalta, Ed. Metaphysics Research Lab, Stanford University, 2016.