CITS3001

Agent Design for The Resistance

Design several resistance agent to beat Random Agent with different possibility

Name: Tahm Zuo

Student ID: 22692786

Table of Content:

1. Preface

The Resistance is an incomplete information game. Members of the resistance do not know who is comrade-in-arms, who is the enemy. The resistance wanted to minimize uncertainty, but the spy knew everything about the battlefield but tried to confuse the resistance by maximizing uncertainty. In the worst case, there are hundreds of different possibilities to consider. A good resistance team member should consider the most likely scenario based on the observed facts, and a good spy should consider the most likely scenario for a resistance team member and make a plan.

2. Abstract

This report first discusses the **potentially usable techniques** for the agent design of The Resistance game (game.py file has been given), which includes *model-based agent architecture* and *learning-based agent architecture*. The report then clearly illustrates **two resistance agents** based on ***model-based*** *agent-oriented architecture* and **attempts to design a resistance agent using MCT**. Finally, the report discussed the **validation** of these three agents and summarized their data when playing with Random spies with several possibilities of betrayal and **reflected** on them.

3. Potentially Usable Agent Architecture

3.1 Model Based Reflex Agent

In The Resistance game, the players always do not have perfect information about the whole game, so the agent has to be able to remember what has happened, like who proposed this mission, who is in the team, if the mission failed or not, how my action affected the game. All this information will maintain a state of the world internally and decide what to do next (Figure 1).

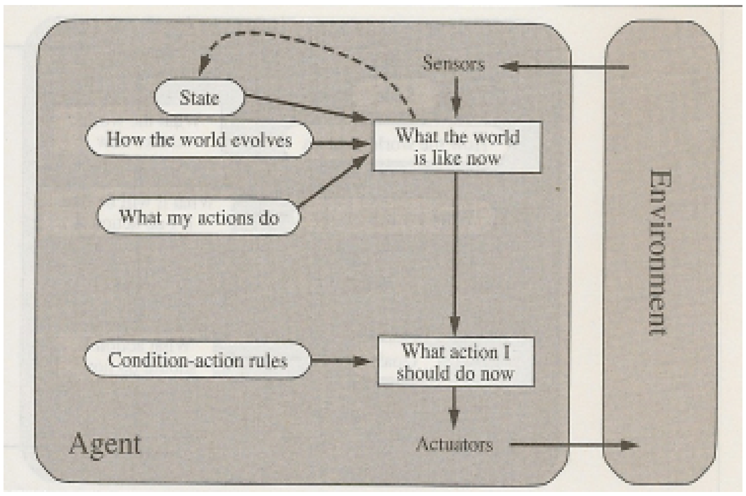
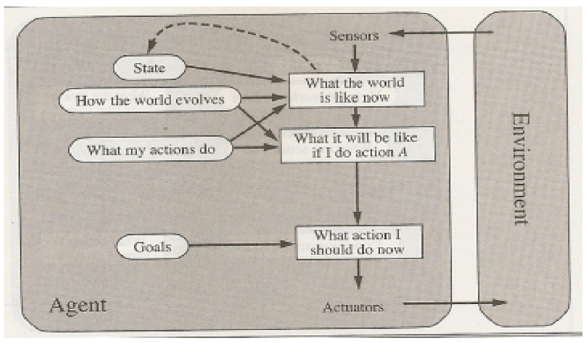


Figure 1: A model-based reflex agent(French, 2021)

3.2 Goal-based Agents/ Utility-based Agents

The Goal-based Agent and Utility-based Agents can usually do better than the Model-Based ones because it not only considers the following activities to do based on the state inside, how the world evolves or what its actions do, also thinks what it will be like if it takes this action(Goal-based) and if it is happy after this. They can be used to write some excellent board game AI (like AI for Go or Chess).

Nevertheless, these two types of Agents may not be suitable for this game because The Resistance is not like a typical board game; the information received by both sides of the game is incomplete and asymmetric. Although a lousy situation for the resistance is good for the spies, the resistance has no way of knowing who the real spies are (and the spies add as much uncertainty as possible by misleading the resistance). So when the agents are playing as a member of the resistance, it can not assume how the world would be changed after this or if they are pleased with this ‘changed state’.


Sensors
State
How the world evolves
What my actions do
What the world is like now
What it will be like if I do action A
Goals
What action I should do now
Agent
Actuators
Environment

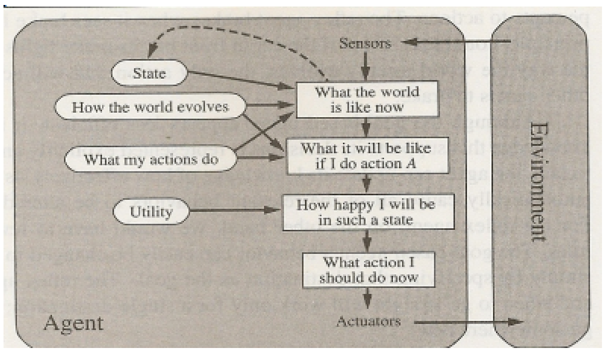

Sensors
State
How the world evolves
What my actions do
What the world is like now
What it will be like if I do action A
Utility
How happy I will be in such a state
What action I should do now
Agent
Actuators
Environment

Figure 2,3 : Goal based/Utility based agents(French, 2021)

3.3 Learning Agents

A general Learning agent has main four elements(Figure 4):

1. Performance element, to choose what action to take; It can shift to a new action later based on the feedback and suggestions from other elements.
2. Critic element determines if the agent's actions make the world better or not and gives feedback.
3. Learning element, taking feedback from Critic element and knowledge from Performance element, trying to figure out how to make a better action next time.
4. Problem generator is concentrating on the development of new experiences for the learning agent to try new actions, which makes the agent keep learning (Learning Agents: Definition, Components & Examples, 2019)

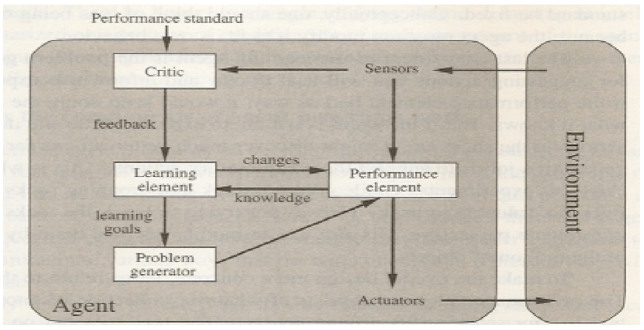


Figure 4: A general learning agent (French, 2021)

3.4 The Theories Could Be Used

3.4.1 Monte Carlo Method

The Monte Carlo method does not find a final, unique result, as the Las Vegas method does, but keeps getting closer to the final result(Taylor, 2014). Like a poll, asking ten Americans whether they approve of Mr Trump might elicit extreme responses (both or none). Nevertheless, when enough people ask, the ratio of those who support it to those who do not tend to converge to a specific value. Alternatively, like the most straightforward coin toss experiment, flip a fair coin and note that it comes up heads or tails. The more I flip the coin, the closer it gets to 1/2. The problem with the Monte Carlo method is that it does not know precisely when to stop. It may depend on the resources it has or as long as the user of the method is happy with the result.

Based on this method, a data structure called Monte Carlo Tree can be introduced. It has the root and the nodes just like a tree. Its core is the iterations that consist of 4 parts: selection, expansion, simulation(figure 1) and backup (Wang, 2021). It is a heuristic search algorithm for specific decision projects and works well in games with large search Spaces. From a global perspective, the main goal is to select the best next step based on the game's current state.

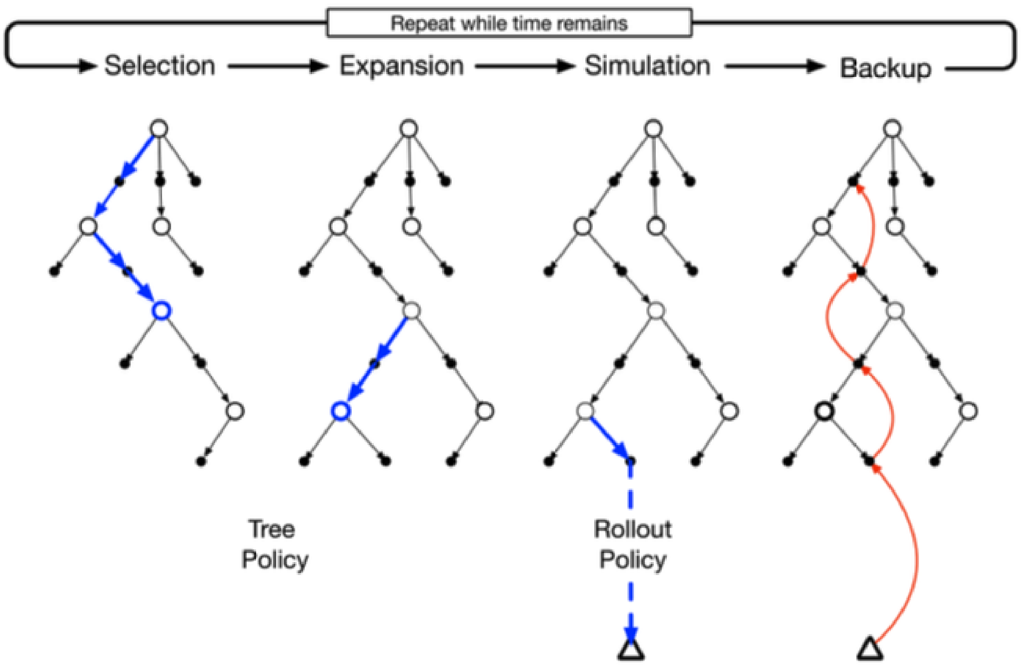


Figure 5: Iteration of Monte Carlo Tree(Wang, 2021)

Although the Resistance game has huge search spaces(the worst case, there are hundreds of different possibilities to consider), Monte Carlo can still handle it as long as it has enough computation resources. The selection of nodes traversing down the tree is achieved by selecting to maximize a certain quantity, similar to the Multiarmed bandit problem, where participants must select a slot machine (bandit) to maximize the estimated payoff for each round. We can use the Upper Confidence Bounds (UCB) formula, which is often used to calculate this formula (shown above), Where v_i is the estimated value of the node (usually calculated by a win rate of the node), n_i is the number of times the node has been accessed, and N is the total number of times its parent node has been accessed. C is an adjustable parameter.

$$v_i + C \times \sqrt{\frac{\ln N}{n_i}}$$

The UCB formula balances the exploitation of known benefits with the encouragement of exploration into relatively unvisited nodes. Revenue estimation is based on random simulation, so nodes must be visited several times to become more reliable. MCTS estimation will be unreliable at the beginning of the search but will eventually converge to a more reliable estimate after sufficient time and reach the optimal estimate in infinite time.

3.4.2 Bayes theorem

Bayes' theorem was developed by the English mathematician Bayes to solve so-called "inverse probability" problems. When dealing with probability in the pre-Bayes era, it was always assumed that heads and tails were equally likely every time you flipped a coin, and then the probability of a particular event (say, ten heads in a row) was discussed under the assumption. Inverse probability looks at things the other way around. For example, if we get ten heads in a row, we want to know the probability of getting heads and tails in a coin flip. The paper on Bayes' theorem was published after Bayes' death, and the great French mathematician Laplace developed the theory into the form we use today, as shown below（Figure 6).

$$P(H \mid D) = \frac{P(H)P(D \mid H)}{P(D)}$$

Figure 6: equation for Bayes theorem

D in the equation represents data; that is, we observe data D in some activity. H represents the hypothesis leading to data D, or it can be intuitively understood as the cause. Several different assumptions can generate the same data D, such as when we leave home in the morning found the current does not rain, but the road is wet, so this may be caused by rain in the morning, also may have been the result of the watering the lawn, is, of course, maybe because the tap water pipeline leak, even alien deliberately put the wet pavement. We also joke about a possibility ("Bayesian learning and future artificial intelligence", 2021).

In The Resistance game, the prior probability cannot be accurately obtained (it is unknown who the spy is and whether the spy will betray the mission), so the agent's cognition is uncertain. Bayes' theorem allows us to start with the facts we already have and use the "subjective probabilities" in our heads to consider them. The prior probability is regarded as a kind of belief, and the premise hypothesis is constantly modified through observation data to form a new belief. Not only that, but after we continue to observe new data, we can continue to use Bayes' principle to integrate existing beliefs and observational data to continuously update the posterior probability and make the "subjective probability" constantly close to the "objective probability".

4. Design of Agents

4.1 improved_Bounder.py (Model Based)

The main idea of Bounder Agent is to trust all players at the beginning of the game; there is a trust_set. When another player fails a mission, it is removed from the trust list. When selecting people to perform tasks, select only players still in the trust_Set (always including themselves). If there are no more players in the trust_set, the player with the fewest failed missions is selected.

Bounder also has a list, clearly_known, for storing identified spies. For example, if Bounder goes on a mission with two other players and fails, and the betrayal count is two, then the two players who went on the mission with Bounder are spies. Bounder will strongly oppose all actions of players already in clearly_known's list.

4.2 Grader.py (Model Based)

The Grader is similar to Bounder's core idea. Raters start the game with a suspicion dictionary called suspicion, and all player numbers are placed in the dictionary as keys, with values being the degree of suspicion of each player. The level of doubt increases when the player who participated in the failed mission, the leader who proposed the mission, and the person who voted for the failed mission. When a mission is successful, the suspicion of the relevant players drops. It also has a clearly_known list, just like the Bounder.

Grader is asked to vote on a team led by someone else; Grader makes a judgment based on the number of spies in the authorities' game. For example, if the number of spies is two, he will object to the actions of the two highest-valued suspicion dict players. When Grader is the leader, the player with the lowest value is selected first, participating in the task itself.

The Grader analyzes more information than Bounder does, such as The person leading The failed task and voting for The failed task. This makes more information available for a more comprehensive analysis of what to do as the game progresses.

4.3 MCT_Agent.py, MCT_decision.py(Monte Carlo Tree)

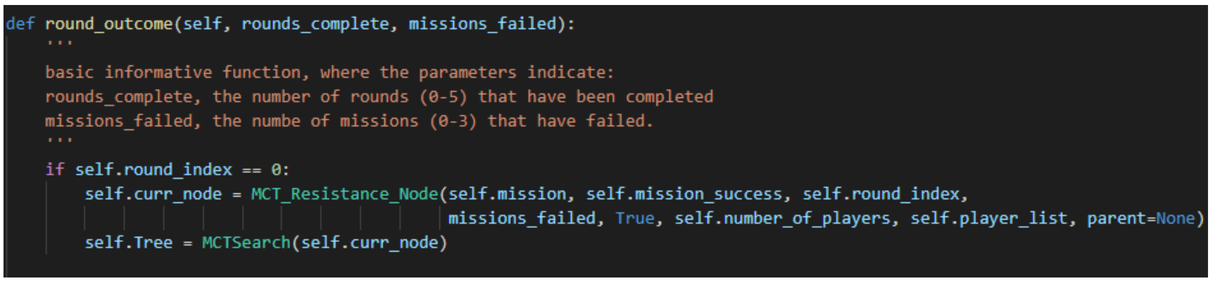
Suppose the first two model-based agents were designed to think like the player. In that case, the implementation of the Monte Carlo Tree results from doing many simulations to cover all possible scenarios to gradually converge to a victory of resistance (i.e., successfully performing three missions).

In the MCT_dicision.py file, there are two types of class: the class of Tree, and the other is Nodes on that Tree. Node instances will have the information for each possible state of each round. In addition to the initialization function, the tree class has two functions: best_action and _tree_class.

The best_action function is the central part that does the four steps of iteration mentioned before. It has an adjustable number of simulation times. This function should return the children of the current node with the highest value after simulation. It has an input value, which is a list that contains all the combinations that have failed in the previous missions. When searching for the best child, it will not calculate the teams in that list.

The _tree_policy function is used to define the basic traversal rule for the Tree, that is, traversal up to the leaf node. In this game, the judgment method of leaf nodes is: If the task fails three times, spy wins; If the number of completed tasks minus the number of failed tasks is greater than or equal to three times, resistance wins.

At the end of the first turn, MCT_Agent creates a unique tree instance for the entire game based on the information from the first turn (Figure 7) as a node and calls the best_action function of the tree instance from that node. Run a thousand simulations to get the highest value for the next round (including which team should be put forward as a leader to participate in the mission and vote yes or no).



```
def round_outcome(self, rounds_complete, missions_failed):
    '''
    basic informative function, where the parameters indicate:
    rounds_complete, the number of rounds (0-5) that have been completed
    missions_failed, the numbe of missions (0-3) that have failed.
    '''
    if self.round_index == 0:
        self.curr_node = MCT_Resistance_Node(self.mission, self.mission_success, self.round_index,
                                             missions_failed, True, self.number_of_players, self.player_list, parent=None)
        self.Tree = MCTSearch(self.curr_node)
```

Figure 7: First generation of Tree and root Node for MCT

After the first round, at the end of each round, MCT will judge the result of the current round and move to the node that conforms to the current round state. With this node as the new root, MCT will continue the simulation for a given number of times and get the node it wants to go to the next round most.

5. Validation of Performances

I changed the selection of spies in game.py so that the new game file Against. Py has the first few agents selected as spies. This makes it possible for all Random Agents to act as spies and all three Agents I developed to act as Resistance. Due to the enormous computational resources required by the MCT agent (tedious instance mapping and hundreds of thousands of simulations), this experiment only tested the data of the five-player game and adjusted the variables by adjusting the probability of spy betrayal in random_agent.py.

The spies of the three groups of experiments consist of two Random agents, while Resistance consists of three agents. A set of data is obtained after 10,000 games of play, with each combination increasing the betrayal rate by 0.2. The data are shown in the following table:

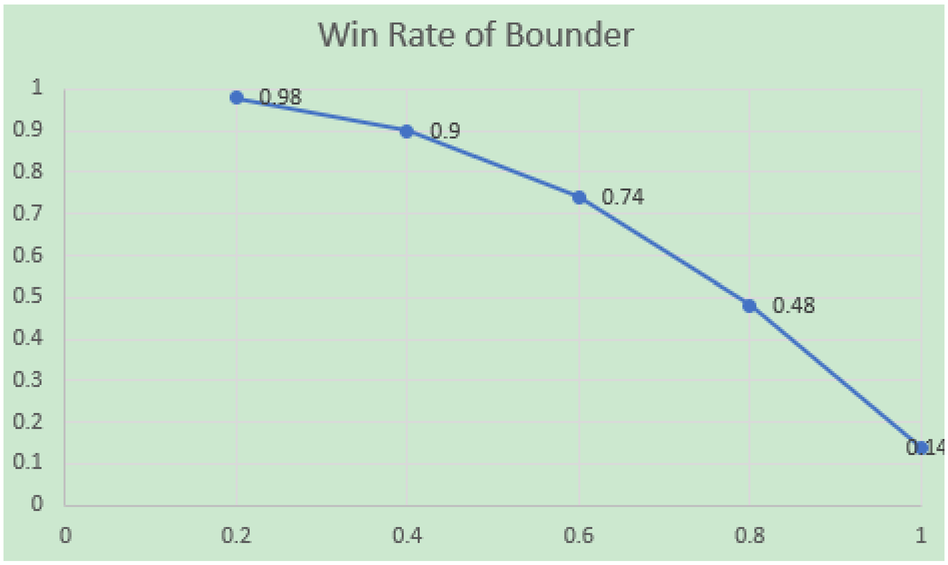


Figure 8: Win Rate of Bounder

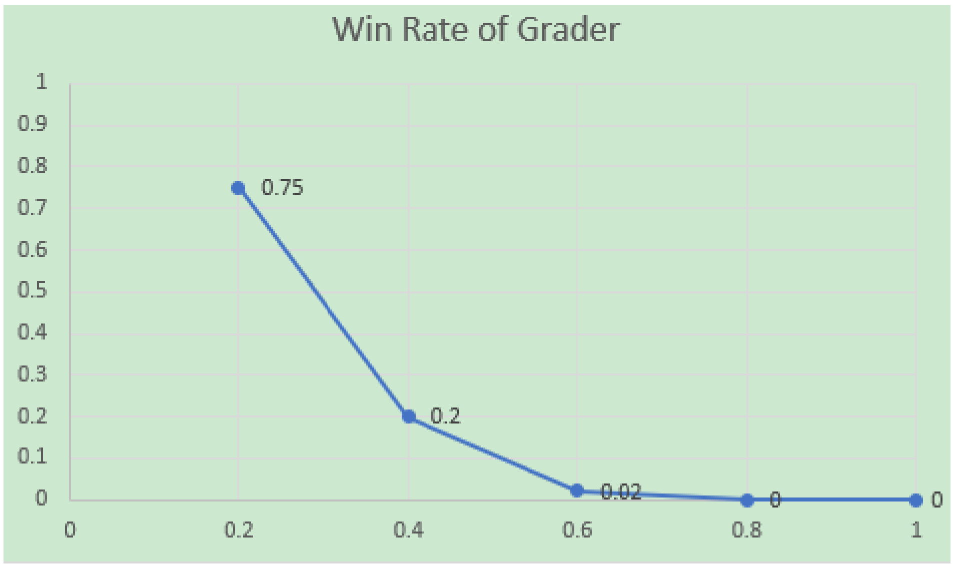


Figure 9: Win Rate of Grader

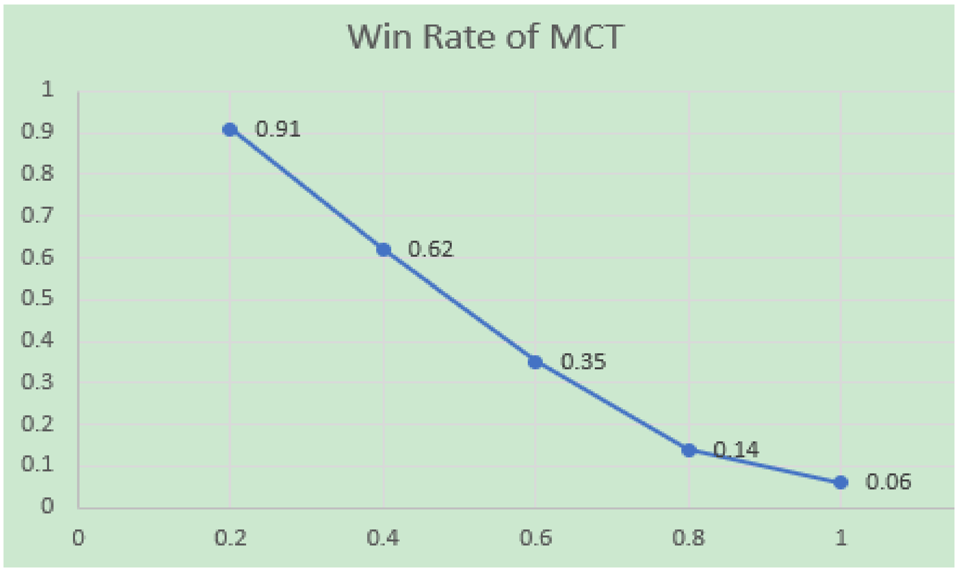


Figure 10: Win Rate of MCT

6. Reflections Based on Validation

As can be seen from the above data, with the increase of the spy betrayal rate, the win rate of the three agents is gradually decreasing and approaching 0. Grader's win rate drops to 0 when the spy defection rate is 0.8 or higher.

For Model-based Agents, this phenomenon is predictable. Because The Resistance, especially for the resistance players, is an imperfect information game. Although Bounder and Grader can make internal judgments as the game progresses based on information such as the outcome of the task and the team members who participated in the task, they do not know if they made the right judgments due to the lack of information. This makes it easy for spies to mislead them into suspecting a teammate who is not a spy.

Monte Carlo Tree's failure, however, was unexpected. Because the Agent realized by the Monte Carlo tree does not simulate the way of thinking of human beings to make the judgment, but carries out more and more simulations with the progress of the situation and selects the optimal situation among all possible simulated situations. Even in the worst-case scenario, MCT Agent should have a 50 per cent chance of winning.

According to a paper called Information Set Monte Carlo Tree Search (Cowling, Powley & Whitehouse, 2012), this situation may be caused by the fact that the state information selection of MCT nodes is not optimal, so the high win rate cannot be achieved through simulation. At the same time, the MCT Agent only learns during each game. Making it remember how it was played and the results (via JSON files, etc.) will both shorten its convergence time and make it more limited (based on past experience, not just authority games).

Reference:

1. Taylor, D. (2014). *A project completed as part of the requirements for the BSc (Hons) Computer Games Programming* [Ebook] (p. 14). Perth: University of Derby School of Computing & Mathematics.

2. Wang, B. (2021). Monte Carlo Tree Search: An Introduction. Retrieved 14 October 2021, from https://towardsdatascience.com/monte-carlo-tree-search-an-introduction-503d8c04e168

3. "Learning Agents: Definition, Components & Examples." Study.com. August 21, 2019. https://study.com/academy/lesson/learning-agents-definition-components-examples.html.

4. Bayesian learning and future artificial intelligence. (2021). Retrieved 14 October 2021, from https://www.jiqizhixin.com/articles/2017-09-19-6

5. Cowling, P., Powley, E., & Whitehouse, D. (2012). Information Set Monte Carlo Tree Search [Ebook] (pp. 120-125). York: University of York. Retrieved from https://pure.york.ac.uk/portal/files/13014166/CowlingPowleyWhitehouse2012.pdf