Fondation in AI project
Werewolf vs Vampire game
Report of Alphagrid team

Philibert de Broglie
philibert.de-broglie@student-cs.fr

Gaël de Léséleuc de Kérouara
gael.de-leseleuc@student-cs.fr

Antonin Duval
antonin.duval@student-cs.fr

1 INTRODUCTION

This project presents the making of an Artificial Intelligence to play the Vampire VS Werewolves game. This game opposes as its name implies it werewolves top vampire, the winning side is the most numerous one at the end. The teams starts with the same number of members. In order to win the game, each team can turn humans into their side or kill some opponents following some specific rules. To be able to create a proper AI and not a random player nor a reflex agent, a global strategy had to be established first, which should take into account the opponents moves.

1.1 Some definitions

- **a unit** is define as member of a team: werewolf, vampire or human
- **an agent** is a group of units which belong to our species will be called an agent independently of its number of units
- **a cell** is where a group of unit is situated. A unit can only move from one cell at a time.

1.2 Implementation overview

To play the Werewolf vs Vampire game, we implement the following features :

- **parallelization** of the simulation : a multi-agent system will organize all the computations
- **expectimax** algorithm with alpha-beta pruning which allow the simulation of complex moves (spliting and combination of multiples units)
- **dynamic heuristic** which depend on the phase of the game

2 MULTI-AGENT SYSTEM

2.1 General overview

The multi-agent systems will :

(1) decompose the map in independent part and proceed each part in parallel
(2) organize the communication between dependent agents by simply organizing a turn by turn system
(3) find the best move of each possible agent by launching the expectimax simulation

2.2 Cluster of independent agents

Independence. Two agents are independents if the distance between each other is greater than the depth max used in the game simulation, meaning that inside the simulation the two groups of units will not be able to directly interact. The idea is that when two groups of units are separated by such a distance, rather than simulating the combination of their moves at each step, their moves can be simulated independently. Hence, the branching factor of the simulations is greatly reduced.

Clustering. When receiving the update message from the server, the agents were clustered into independent group of agents, so that two agents which belong to different clusters were viewed independent. Hence, each cluster represent one unique region of the game.

Thread generation. For each cluster of agent, a new thread is launched; each thread having two seconds to return all best moves of all agent belonging to the cluster.
At the end, all the moves return by each thread are concatenated.

2.3 Management of dependent agents

Combinations generation. The first idea was to consider a group of dependent agents as a single entity. So at each step of the simulation, all the possible combinations moves for an agent are simulated. Problem was that the branching factor was so high that it was not possible to manage more than two dependent agents.

Turn by turn system. To overcome these limitations, the simulation was simplified thanks to the implementation of a turn by turn system. The idea is the following :

(1) considering a group of dependent agents, choose arbitrarily one agent
(2) run a simulation for this agent and inside this simulation "freeze" all the other agents (to not simulate the other agent moves for now)
(3) select the best move for this particular agent
(4) update the grid by applying the selected best move and repeat with another agent

The inconvenient of this approach is that it may not find the best possible moves because it is dependent on the order of agent which play first. However with this system, come a great advantage : combination computation was suppressed, hence the branching factor of each simulation was lower and thus the algorithm got exponentially faster at explore the remaining possible choices.

Comparison of the two systems. So the question was, is it better to :

- be able to manage fewer agents while being able to find the optimal combination
- be able to manage a lots of agents but to generate moves that are less optimal.

To solve this issue, both solutions were implemented and launched against each other. It turned out that the "turn by turn system" was winning against the "combination generation system". Mostly

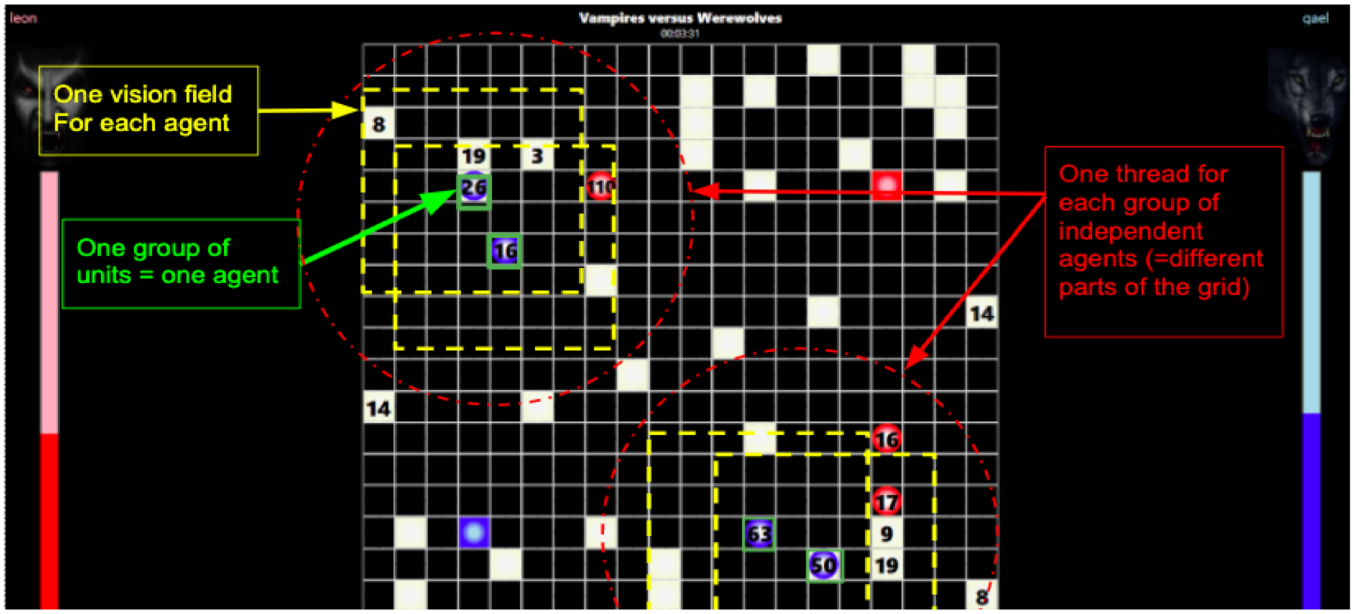
leon
Vampires versus Werewolves
00:03:31
qael
One vision field For each agent
One group of units = one agent
One thread for each group of independent agents (=different parts of the grid)
8
19
3
26
110
16
14
14
16
17
63
9
50
19
8

Figure 1

because it was able to handle a lot more group of units and thus faster at killing human.

3 EXPECTIMAX WITH ALPHA/BETA PRUNING

3.1 Generate move

To generate the list of moves one agent can do, a function **generate_move()** that take as parameters the grid, the position of the unit to move, and the species of this unit, was created. There is a two case scenario :

- Generating a move for the enemy.
- Generating a move for us.

Two scenarios had to be differentiated as the turn-by-turn system for is only used for our units. Therefore, all combinations of moves the enemy could do with his group of units had to be computed. When simulating a move for our player, it was not needed to compute the combination of moves of different agents. So let's explain how each scenario was handled :

Generating a move for us :

(1) Given the agent position, test every move and store it if it's a possible move (moves that are off grid and unauthorized are deleted).
(2) Add all combinations for splitting in two the group of unit. This mean only two group of the same size can be generated, but this reduces the computation's time tremendously.

Generating a move for the enemy :

(1) For each enemy positions, test if it's in the vision field of this agent(see 3.4), and if true, continue.
(2) test every move and store it if it's a possible move (moves that are off grid and unauthorized are deleted).
(3) Add all combinations for splitting in two the group of unit.
(4) Add all combinations of moves between all groups of enemy that are in sight. This step can create a lot of possible combinations. This is why it is only used for enemies that are in a danger zone.

3.2 Sort move

Since lots of different moves are being simulated, it is important to use a sorting system in order to simulate the best move first. This gives two advantages :

- In case of multiples moves having the same heuristic, our agent can be forced to choose a certain one.
- When time runs out during computation, the best moves were simulated first.

The sorting system used is actually very simple : every move are applied to the grid and their heuristics (cf 4.1) computed. Using this metric, the moves can be sorted accordingly to the species that has to be simulated.

3.3 Fight simulation (expectimax node)

When testing a move, the situation where to attack a group of unit will come up, and it includes some random outcome in the simulation, especially with the random battle scenarios. Every time a move that generate a fight is tested, those moves are stored in a list. If this concerns a deterministic fight (the outcome being always the same), then the new grid is computed and the heuristic is returned. If it's a random fight scenario, the two outcomes (victory and defeat) are stored with their according probability.

In order to have the correct expected value for one move including multiple fights, all combinations of possible results (For n fights, there are 2^n combinations) are computed.V_i and D_i are . notes as the state of victory or defeat for a fight i. If two fight happens at one move, this will give : (V_1 and D_2) or (D_1 and D_2) or (D_1 and V_2) etc...

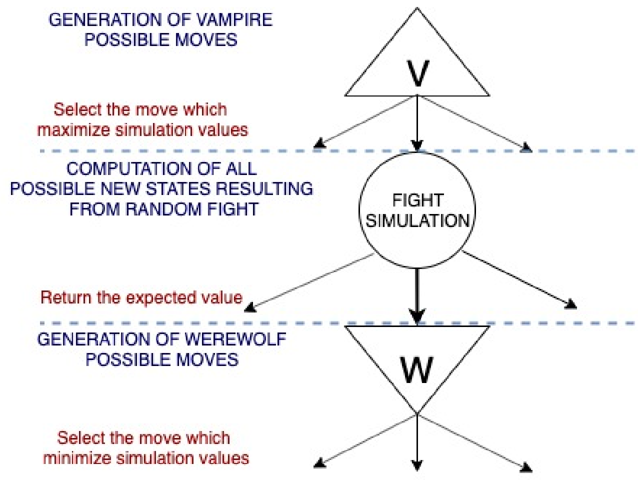
GENERATION OF VAMPIRE POSSIBLE MOVES
V
Select the move which maximize simulation values
COMPUTATION OF ALL POSSIBLE NEW STATES RESULTING FROM RANDOM FIGHT
FIGHT SIMULATION
Return the expected value
GENERATION OF WEREWOLF POSSIBLE MOVES
W
Select the move which minimize simulation values

Figure 2: Handling randomness in our simulation

Let's denote m the move to simulate, $S(m)$ the expected value for move m, p_i the probability of combination i,$S(i)$ the simulation value for state i, and Δ all the combinations computed. This gives us the following :

$$S(m) = \sum_{i \in \Delta} p_i * S(i) \tag{1}$$

This allows us to have a value for the heuristic that takes into account the probability of each state after a fight.

3.4 Vision field

Since a lot of moves and many combinations our to be computed, it can become quite large. it was decided that limiting the view of the agents could be useful in that case. Indeed, one agent does not need to see what the enemy does if it is out of his reach.

The notion of **reach**, express what is the furthest point where an enemy can attack an agent. This is actually *depth max*. Indeed, for a depth 3, the maximum distance our agent can move during the simulation is 2 cells, and the enemy agent 1 cell. Therefore, the vision of an agent can be limited to only depth max. If an enemy is out of the vision field, as you can see in figure 1, his moves will not be computed in the expectimax. Instead, more moves for an agent could then be simulated.

3.5 Time control

There is a limit of 2 seconds for sending a new move, so the computations has to end before time runs out, even if not all the nodes were explored. To deal with this, all agents were given a time limit to do their computation, equals to :

$$t(in\ seconds) = \frac{2}{N} - 0.01 \tag{2}$$

where N is the number of agents. When the agent reach this time limit, he must stop simulating new moves and return the best answer found yet.

4 HEURISTIC

4.1 Four factors

General overview. Our heuristic is composed of four factors : win-factor, split-factor, kill-factor, merge-factor. Such as our final heuristic is :

$$\mathcal{H} = \alpha \cdot win_f + \beta \cdot split_f + \gamma \cdot kill_f + \delta \cdot merge_f$$

Win factor. the win factor is simply to indicate which species is currently winning.

$$win_f = N_v - N_w$$

where N_v is the number of vampires and N_w is the number of werewolf left on the grid.

Split factor. to compute the split factor, first was browsed the position of each group of humans (h) and then was determine if this group could be killed first by vampires (then it belong to ensemble we call H_v) or by werewolfs with probability one (in H_w). It was denoted $d(h, v/w)$ the distance between the group of humans and the closest group of vampire/werewolfs that can eat them with probability 1. With this notation, the split factor was computed as :

$$split_f = \sum_{h \in H_v} \frac{N_h}{dist(h, v)} - \sum_{h \in H_w} \frac{N_h}{dist(h, w)}$$

Concretely, this mean it is preferable to have a lot of possibility to kill humans and also to be close to humans that can be killed. This was call the split factor as this heuristic enhance the split when multiple group of humans are in opposite directions.

Merge factor. As some point during the game, gathering our units may be a strategic move.Hence, a merge factor was brought up and defined as:

$$merge_f = -\frac{\sum_{v_i, v_j \in V} d(v_i, v_j)}{\text{nb of vampire group}} + \frac{\sum_{w_i, w_j \in W} d(w_i, w_j)}{\text{nb of werewolf group}}$$

Kill factor. Finally, the last idea was to include the fact that it is better to be close to enemies which can be killed and far away from enemies that can kill us.

$$kill_f = \sum_{v \in V, w \in W} \frac{n_v - n_w}{dist(v, w)}$$

4.2 Dynamic

Then, these factors were dynamically weighted depending on the game phase.

α - win factor. Concerning the win factor, the α weight was maintained constant over the entire game

β - kill factor. β was set equal to $\frac{1}{\text{number of humans}}$ so that at first when there is an important number of humans on the map, our unit focuses on killing humans rather than on killing enemies, fewer humans are left, the more aggressive the agent get.

γ - merge factor. γ was set equal to $\frac{1}{\text{number of human groups}}$ so that our agents don't merge when there are many groups of human; but when there is no more, they try to merge as soon as possible. Once there is no more humans on the map and that the agents forms only one single group of units, γ is set to zero.

δ - split factor. δ , was simply set to zero when no more humans were on the map.

5 LIMITATIONS

5.1 Results

Even though we manage to create an algorithm that can play the game fairly well on the map we've created, it was very disappointing for us, to see our AI act so poorly the day of the final challenge. Our algorithm crashed multiple times, unexpectedly. This is mainly due to the fact that not all possible scenarios were considered and that the code was not flexible enough to handle every kind of map. The major bug which was faced was that there were no maximum nor minimum value for alpha and beta large enough, thus the algorithm ended up with scenarios where all moves were pruned without selecting any, resulting in an empty list of moves.

However, when the code finally worked, probably thanks to maps it could handle better, it was a relief to observe that our algorithm could won games against all the other AIs.

5.2 Limitation of the minimax

One of the downside of using minimax is that it suppose the enemy will behave the same way as we do. However, as observe during the competition, this is often not the case. For example, our heuristic was made in a way that killing humans is a priority. Getting closer to an enemy is not a problem as it is thought that the enemy will not attack us and also prefer eating humans. When facing an AI that is more aggressive, we might end up loosing because of how we expected him to act.