Project Report
Vampires versus Werewolves
CentraleSupélec
Foundations of Artificial Intelligence

Alexandre Duval, Leo Fillioux, Sebastien Saubert

18 December 2019

Introduction

The goal of this project is to develop an Artificial Intelligence to play the Vampire vs Werewolves game, according to the set of pre-defined rules. Before delving into the details of our AI, we would like to mention the main ideas we have tried to implement throughout this project. Some of the key steps we wanted to take were:

- communicate effectively with the server and play within 2 seconds no matter what
- use minimax with alpha beta pruning
- fine tune the heuristic. Take into consideration enemies, eatable humans, random battles, distances...
- create a SmartScan that would give us long term thinking and restrict the number of possible moves (only interesting moves)
- find ways to store some results and win computation time
- test on many different maps to make sure our player is robust. As a result, no bug was encountered during the competition which is a great achievement.

1 Progression

We decided to address the project by iterating over different versions of players starting from a simple one to more complex version by adding functionalities.

At first, we developed our **Random player** which generates moves randomly with no depth consideration. Concretely, from a given cell, we have 8 options (except if we are on the edge of the grid) and we equally pick one of these options to decide the move to perform. What is the benefit of this player? As it is our first player, it allows us to manage more easily the technical part and especially the communication to the server with sockets: send move(s), receive update(s)... As the player code was simple, debugging of the communication was easier to perform. With this task done, we knew that our communication interface with the server was complete and behind us so we can now focus and more advanced players.

Then, we created the **Conservative player** which is a great leap forward from the random player as Minimax with alpha beta pruning (see section 3), and so d-depth tree exploration was implemented. This player was called conservative as it doesn't split and only targets humans or enemies that are eatable, and so, does not engage in any random battle.

We started to face some performance issues with Minimax as the branching options where exploding with the depth ($\approx 8^{d-1}$). On top of that, the conservative player strategy was too protective. We created then the **Smart Greedy player** that allowed us to manage random battles and implement the SmartScan (see section 5) that makes the player naturally more greedy.

We then performed a change in the computation of the Minimax heuristic by using a weighted sum of the SmartScan results (see section 5). We called this player **Smart Player**. We also implemented a clock to manage the time constraint (see section 4).

At this point, we have a robust player but without split. Clearly, this is the next great leap of our players' journey. We created the **Smart Split Player** that integrated the split in move generation (see section 2). We also implement cache management to store intermediate calculations (see section 4).

At last, we developed our last version of our player: the **Merge player**. Indeed, with the split, the player was splitting but fails to merge back quickly and easily especially at final stages of the game. To mitigate this, we integrated bonus at end game for *mergers* in the SmartScan (see section 5).

In the end, we can see that we had 5 different versions of players. Should we have gone directly with the *Merge player*? We don't think so. Indeed, iterating over different versions allowed us to better understand the dynamics of the game. It also eases our testing by considering only one or two new functionalities at a given iteration. We faced many bugs at each step and performed many tries regarding the heuristic. If we had done it entirely at once, there is a big chance that the player would have had many bugs whose root cause is not easily identifiable. On top of that, as we have many players, we were able to make them fight each other to ensure the change of behaviour and the continuity in the code or strategy.

2 Moves generation

At each step, we want to generate the possible moves for each group of players. This is composed of two parts. The first one is the non-split options, which represent the moves where the group does not split, i.e. moves all the units to the same position. There is therefore a maximum of 8 possible positions a group can move to. For example, the green units in the following example has the following possible moves :

- move m_1 : move 8 players to (0, 1)
- move m_2 : move 8 players to (1, 1)
- move m_3 : move 8 players to (1, 0)

The second type of moves which can be generated are those in which the group of units will split.

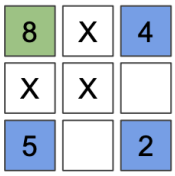

Figure 1: Non split options

One group can only be split into two groups at each turn, and we limit the total number of groups to 4. To generate the splitting options, we look at the 'eatable humans' at a distance 2 of the considered group (by 'eatable humans', we mean the groups that we are sure to eat). In this list of 'eatable humans', we look at each pair and only keep those where the sum is still eatable : this represents the case where we can split into two groups that will eat two groups of humans. Each cell that we chose is called a target cell. The group is split into the cells leading to the target cells, i.e. the cells next to our unit that lead to the target cell. The number of units in each split is determined in the following manner :

- start by moving the same number of units as in the target cell
- if there are some units left, start by adding them to the smallest of both cells (until they are equal), this is because we want to avoid having very small groups that are quite vulnerable
- then add equally to each cell

For example, in the previous situation, we would generate the two following splitting options :

- move m_4 : move 5 players to (0, 1) and 2 players to (1, 1), this is then evened out to (5 and 3)
- move m_5 : move 4 players to (1, 0) and 2 players to (1, 1), this is then evened out to (4 and 4)

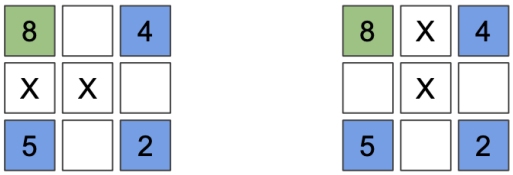

Figure 2: Non split options

We concatenate the possible moves for each group of units. For example, for this group of units, the possible moves would be $\{m_1, m_2, m_3, m_4, m_5\}$. Before returning the moves, we make sure that the target positions are not a previous position, because a cell can not be a source and a destination.

We do this for each group of units of the current team on the board. To generate the moves than will be analyzed by the SmartScan and minimax, we take the combination of the possible moves of each group, which represents the possible boards after the current turn. For example, if there had been another group with 2 possible moves $\{m_6, m_7\}$. The possible boards after this turn are the ones that occur after these moves $\{(m_1, m_6), (m_2, m_6), (m_3, m_6), (m_4, m_6), (m_5, m_6), (m_1, m_7), (m_2, m_7),$

(m_3, m_7), (m_4, m_7), $(m_5, m_7)\}$.

In the next section, we will see how these available moves allow us to develop a tree that will be applied to the Minimax algorithm to identify the best move to perform.

3 Minimax and alpha-beta pruning

After the implementation of our first random player, we decided directly to move on to a player that can simulate plays on depths greater than one. The obvious choice was to implement Minimax with Alpha Beta pruning. We will not develop this algorithm as it is well describe on many online sources[1]. On the other hand, for Minimax to work, we need to have a clear definition of the following concepts: game representation, branches exploration & recursivity & tree-depth management and heuristic computation.

3.1 Board representation

To be able to play the game, we need to have at each time the corresponding information:

- A board of size w and h (do not change within a game)
- Humans units located on a position (x_h, y_h)
- Werewolves units located on a position (x_w, y_w)
- Vampires units located on a position (x_v, y_v)
- What type of unit we are controlling (Werewolves or vampires - does not change within a game)

To manage this information, we created a class ***Board***. For the humans, werewolves and vampires units, we decided to manage the position with 3 dictionaries where the key is the position of the unit and the values the number of creatures of the unit. As the board is 'sparse' from humans, werewolves and vampires units, we think this is the best representation for memory usage and board scanning. Finally, this class board manage a static representation of the game.

3.2 Branches exploration, recursivity and depth management

To be able to explore a game situation and decide the best move to play, we need to explore our tree t of depth d with Minimax. From our initial *board* b_0, we compute new boards b_{1m} by integrating to b_0 a move m (see section 2):

$$b_{1m} = b_0 + m$$

Concretely, we integrated to our class *board* a method that, for a given move, simulates the play (move on empty case, fight versus humans or enemies) and return a new instance of board. As Minimax was implemented recursively, in fact, the method performed the following calculations :

$$b_{dm} = b_{d-1} + m$$

The last thing to consider is to control the depth of the tree. We simply add to the Minimax recursive function a depth argument that starts at the required depth d and decrease by 1 at each

[1] https://en.wikipedia.org/wiki/Alpha–beta_pruning

recursive call. When the depth reaches 1, it indicates that we have reached the leaves of the tree t.

3.3 Heuristic computation

Heuristic is calculated for each leaves of the tree t (when the depth $d = 1$) except when alpha beta pruning generates some cuts. For all of our players' version, the heuristic is calculated **only our player's point of view**. This means that we can't allow the leaves to be game states of our enemy. To control that, we ensure that we run our players and Minimax with an odd number of *depth to explore* so that the leaves represent our boards.

Our first players computes a heuristic which was a function of:

- Humans units and distance we are from them
- Enemies units and distance we are from them
- Total amount of our units versus the total amount of our enemy's units

We will not go deeper in the formula as this time as the time constraint of the game (see *section 4*) and the implementation of **SmartScan** clearly redefined our calculation of our heuristic (see *section 5*)

4 Time constraints

At each step of the game, there are a lot of possible outcomes to consider (as we have seen in the previous sections), the number of which explodes as we consider more split possibilities and more enemies. This takes a considerable amount of time to evaluate and because we want to look as far ahead in the game as possible, we will try to save as much time as possible.

4.1 Cache

One great way to save time is to avoid doing the same calculations multiple times. One way to do this is to cache the results of the calculations into a dictionary. There is one dictionary per type of calculation, for example when evaluating the score of a group of enemies. This dictionary is reset either at every round, or at every level of the minimax algorithm. The dictionary uses as keys either :

- a hash of the position taken into consideration : in this case just a tuple (x, y, nb_units)
- a hash of the board, which was a bit harder to put into place : we used a text concatenation of all the (x, y, nb_units) on the board with humans, vampires and werewolves, this ensured that the hash would be unique and would be the same if the board was the same

The value associated to this key if the result of the calculation. This way, before doing a calculation, we check if the key (i.e. hash) is already in the dictionary. If it is, we don't reevaluate, otherwise, we run the calculation and update the dictionary. This allowed us to avoid doing some computationally expensive tasks more times than necessary.

4.2 Clock

One turn can not take more than 2 seconds, otherwise the player loses its turn and can result to a game loss if not well managed. Without any restriction, depending on the board, the calculations for one turn can range from a few tenths of a second, to more than ten seconds. We want to make sure, than even if the board is supposed to take a very long time to evaluate, that we will communicate our moves to the server in time. For this, we decided to implement a **Clock** class which is initialized at the beginning of each turn. Before each calculation, we check if the time elapsed since the initialization is greater than a certain constant. For the 2 second time limit, we set this constant to 1.95 seconds, to have some margin. If we are under the 1.95 second limit, we go on with the calculation, otherwise we end every calculation and return our best result yet.

5 SmartScan full strategy

As mentioned in the introduction, the SmartScan was originally uniquely designed to reduce the number of possible moves we could take, disregarding non-relevant moves. In the final version, it is probably the most important part of the AI since it is used at every level of the tree and in the heuristic to determine what move we should take next. But what is the SmartScan? It is simply a function that attributes a score to every possible move. It thus allows us to consider only the k best moves in the tree search, which allows us to save precious computation time. This function has access to the full board (at the present state) and acts as a general intelligence. It is a bit greedy and incorporates long term thinking. Each potential move is rated based on a series of factors. More precisely, different incentives are given for some carefully targeted achievements. Since we score a unique position (x, y, nb_units), you may wonder how this function deals with the split? In a word, it sums the SmartScan score of each unit for a given 'global' move.

$$\textbf{\textit{score}} = humans\_and\_enemies + eatable\_humans\_proximity + enemies + merge + end\_of\_game$$

Let's now delve into the details of this function.

- We consider **humans and enemies** that happen to be in the cell we move to. We would like our bot to be a bit greedy and give him an incentive to attack humans and enemies that we can eat for certain. The incentive to eat enemies is relatively higher than those to eat humans. Regarding enemies that we cannot eat for sure, we avoid big ones (1.5 greater than us) and are relatively indifferent to other sizes. We make our bot slightly aggressive, meaning there is a small incentive to attack enemies smaller than us and slightly scared of those bigger than us; weighted by the difference of units in both cases. This is conformed to the random battles outcomes probabilities.

- We then consider **the number of eatable humans and their proximity**, weighted by the probability to eat this group. The number of humans is denoted by n, the proximity d refers to the distance separating our unit to a group of human and the probability to eat this group is defined by pq, where p is the probability to eat them given our potential number of units and q the probability to eat them before the opponent. Note that given the move regarded, we split the board accordingly and focus only on the part of the board that is relevant to the direction followed with this move, meaning the part of the board corresponding to the direction we are heading to (schema power point presentation). This

makes us save computation time and is as relevant as looking at the whole board. Coming back to our incentive formula, we estimate our potential number of units when reaching a human group by $N + \sum_{i \in E} h_i$ (N is our initial number of units and h_i the number of humans closer to us than the human group considered, which we determined to be eatable previously). It thus has to be computed iteratively, looking first at humans closest to our position and augmenting our estimated number of units accordingly. On the other hand, the value of q depends on the proximity of the closest enemy to that human group. q = 0.5 when a human group is much closer to the enemy than us and 1 when much closer to us. If the human group is in between, we set q=1.2 to encourage our AI to eat 'disputable' humans, which often allows us to take a non negligible competitive advantage. In conclusion, for each potential move, the incentive added is $\sum \frac{n}{d+1} \cdot p \cdot q$.

- Next step is obviously to consider **enemies** that we will not eat straight away. We tackle various cases and give an appropriate incentive. We adopt a similar behaviour to the one described in the enemies paragraph above. We also strongly encourage our bot not to go within a distance of 1 from an enemy much bigger than us, as this would probably result in a certain death.

- Since our bot now splits well and plays well, what is left to do is considering **merging**, which is a key aspect of the game. It is indeed extremely important to merge well to win the game. We encourage merging by minimising the distance between our units. This is created on top of our previous strategies, so we will still eat humans if they are on the way and avoid big enemies. We give merge incentive in several cases:

  - number of humans left + number enemies < number of units (medium incentive) : if the enemy has no chance to win without attacking, we would like to merge quickly to avoid taking any risk of losing by letting a sub-unit be attacked

  - no humans left (strong incentive) : the fate of the game will probably result in a battle and we want to maximise our chances of success

  - a few humans left (small incentive) : in case the humans left are difficult to eat (given distance of enemies position), we would like our AI to merge and go kill the opponent before he can develop merging strategies, which we assume he will do when no humans are left

- Finally, we specify **end of game strategies**. We decide to attack enemies if they are 1.5 less than us or more than us. We run away if we are a bit more because we are winning and do not want to have a chance to loose by launching a random battle. The last case taken into consideration is if the enemy is split. Here, we choose to attack the smallest closest enemy unit.

- A big **optimisation** part took place all along the process to scale the incentives we give, in order to obtain the desired behaviour form our AI.

In conclusion, creating this SmartScan helped us greatly improve our bot. Its complexity and long term view helped us to play cleverly. Because of the time constraint of the game, reducing the number of branches to be explored in TreeSearch and sorting them from the most relevant to the

least turned out to be a key asset. Also, as evoked earlier, SmartScan enabled us to deal with the exploding computation time related to the heuristic calculation. Indeed, using as heuristic a weighted sum of the SmartScan (computed from the root to the leave) turned out to be not only faster but also more relevant.

$$Heuristic = \sum_{d \in odd\ depth} SmartScan_d$$

Note that only the SmartScan corresponding to an odd depth in the TreeSearch were considered as they correspond to our moves (not the enemy).

Conclusion

This project was a great opportunity for us to get hands-on experience with the algorithms we have studied in class and to better understand how they work. We decided to start with a very simple player and added more and more complex architectures and algorithms. Even though not all of them ended up improving the performance of the player, they helped us to understand what was needed for our AI to perform well in certain situations. The time constraint was a good way for us to find tricks to avoid useless calculations and to treat time as a valuable resource that should be spent wisely on calculations.

Overall, we were able to create a good performing AI with a Minimax depth of 5 and were very happy with its performance. While its behavior was not entirely optimal in some situations, it took mostly smart decisions and adapted well to various types of maps. On top of that, it had no bugs and ran smoothly. To make sure of this, once we had build an AI with which we were satisfied, we specified a dozen of situations and the way we wanted it to react. We then made sure to tune the hyperparameters so that the specifications were satisfied. We therefore had a model which had a solid general behavior and which reacted well in particular situations, such as when we expect the groups to merge once we are towards the end of the game.

Nevertheless, we still had room for improvement as the behavior was sometimes unsatisfactory. We mainly felt that the player was a bit too aggressive, which was designed on purpose but often played to our disadvantage given the superiority of our AI on others. In the competition, in the map where both players start back to back, ours chose to attack immediately, which was perhaps not the best option. We also felt that the performance of our player would have been improved if we adapted to different types of maps. For example, by augmenting the depth of the Minimax when calculations were light enough (i.e. small maps, low number of humans, ...).