DeepBlue

Marco Farfan, Nabil Mouadden, Yuchen Jiang, Paul Dampierre, Alassane Watt

February 2021

Introduction

In this project we intend to implement an artificial intelligence capable of playing the Vampires VS Werewolves game. The point of the game is to kill all the opponent creatures and therefore be the dominant species. To kill the opponent's monsters, we can move to the same cell as the opponent player and provoke an attack. If two opponent factions are in the same cell, an attack occurs and the faction with the most monsters is more likely to win. To maximize our chances of winning, we can convert humans to our monster faction by moving to their cell. The game is based on turns, at each turn the player must perform at least one move and can move to any adjacent cell. Our project is to run an algorithm to perform the decision on the cell movements to maximize our chances of winning the game.
To perform such a task, we had to break down the project into sizeable chunks. The first part was to understand the game and write a skeleton code to communicate with the server. The second part consisted of formulating a strategy and an algorithm to win the game. Finally, the last part was the testing and analysis phase of the project in which we studied the behavior of our AI and optimized it to maximize our chances of victory.

Rules

Six main rules must be considered in this game:

Rule 1. At least one movement by play. This means that if we have many groups of monsters of the same kind, we have to move at least one of such groups. This movement could be either partial or complete.
Rule 2. You can only move your species group. This means that it is not allowed to move an enemy's group of monsters during your turn.
Rule 3. You need to have enough creatures on a cell to perform all the moves from this cell. This implies that the moves of a cell can be performed as long as you have creatures in that cell.
Rule 4. You can move in 8 directions (unless on borders) as described in figure 1.

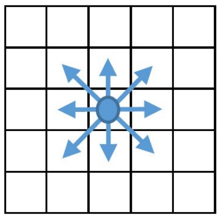


Figure 1: Potential Movements

Obviously, if you are positioned on a border, the number of possible moves reduces.
Rule 5. In the same ply, a cell cannot be target and source of moves. This means that when moving two or more monsters we cannot make the destiny position of one monster the same as the origin position of another in the same play.
Rule 6. You have to move at least one creature. In other words, a move of a group of monsters has to involve at least one creature and at most the number of creatures of that group.

There are two species in the competition and several humans on the board. Our goal is to become the dominant species. To do that, we can convert humans into our species to enhance our population or kill enemy monsters. To convert a group of humans, we have to be as numerous as them, otherwise a random battle starts. Similarly, to kill a group of enemies we have to be 1.5 times greater than them, if they are 1.5 times greater than us, they will kill us, otherwise a random battle starts.
Let E1 and E2 be the number of the two species (opponents or humans) and E1 be the number of monsters attacking:

$$\text{if E1} = \text{E2}, \text{ then P} = 0.5$$
$$\text{if E1} < \text{E2}, \text{ then P} = \frac{\text{E1}}{2 \times \text{E2}}$$
$$\text{otherwise, P} = \frac{\text{E1}}{\text{E2}} - 0.5$$

The probability that E1 wins is given by P.

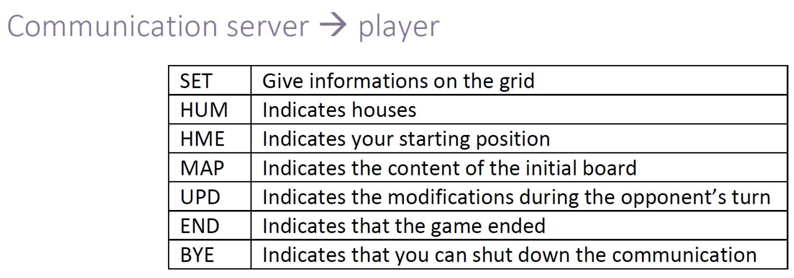
Additionally:
- If the attackers win, each of them has a probability P to stay alive. Moreover, if they win over humans, each human has a probability P to survive and to be converted.
- If the attackers lose, each enemy has a probability 1-P to stay alive.

Finally, there is a time constraint. We only have two seconds at each turn to play. If we are not able to move our creatures in that time, the server will pass the turn to the other player.

Server Communication

To play this game, our AI has to communicate with the server (bidirectional communication). We connect via a TCP Socket. There are many commands coded on 3 bytes that we can use to send data to the server or receive data from it as can be seen in figures 2 and 3. We secure the communication with the server at the beginning of our program, in the initialization module of our class Game.

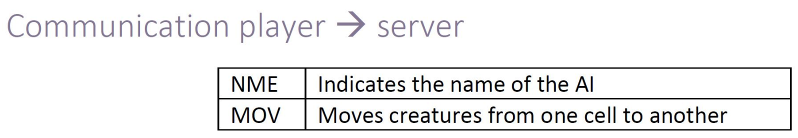
We also define a function “receive_data” to read the data in bytes that comes from the server and unpack it. In this section we define the name of our monster, we read the number of rows and columns of the map, we read our initial position coordinates, information about the humans, among other things. Additionally, based on our initial position we define a variable containing our relative position so that we do not need to worry about being player 1 (vampires) or player 2 (werewolves), our program will run smoothly in either case.



Communication server → player

SET	Give informations on the grid
HUM	Indicates houses
HME	Indicates your starting position
MAP	Indicates the content of the initial board
UPD	Indicates the modifications during the opponent’s turn
END	Indicates that the game ended
BYE	Indicates that you can shut down the communication

Figure 2: Communication Server



Communication player → server

NME	Indicates the name of the AI
MOV	Moves creatures from one cell to another

Figure 3: Communication Player

Game Board

In order to follow up changes in the board at each turn, we need to first define our board, then initialize it and update it at each play. To this end, we defined a “create_board” function that based on the height and width of the map, it creates a board variable (a multidimensional array) that will display the current state of the board game at each turn. It will basically have 3 dimensions: the row, the column and an array of size 3 at each position to indicate the triplet (Number of Humans, Humber of Vampires, Number of Werewolves). In addition, this function initializes our board variable with the data obtained from the command MAP in the server communication module. We also define a “update_board” function that updates our board variable based on last moves in the map. To do that, it first uses the UPD command to read our last move and the opponent’s move. Then, using this information we update our board array accordingly. Finally, we created a “print_board” function in order to monitor the state of the game at any time. This function will simply print our board in a nice format (height x width x position).

Distances in the Board

We defined a function “calculateDistance” as our simplest method to determine how far two cells are from each other. Therefore, this function computes the Euclidean distance between two points in a 2D space.
However, this is not a very precise way to compute distances in a game with moving elements.

Hence, we defined another function, "calculateDistance_plays" that computes the distance between two given coordinates but in terms of number of plays. That is, in terms of the number of moves required to get from the original point to the target position. We deem this is a more suitable way to compute distances in this game. For example, in figure 4 we can see that using the function "calculateDistance" the vampire (red) group of monsters would choose B over A as the next move in terms of closeness to the group of humans (white) because B is 1 unit away from the cell containing the humans, whereas A is $\sqrt{2}$ units away (1.41). In contrast, "calculateDistance_plays" would consider both options equally favorable in terms of closeness to humans because both are 1 move or play away from them.

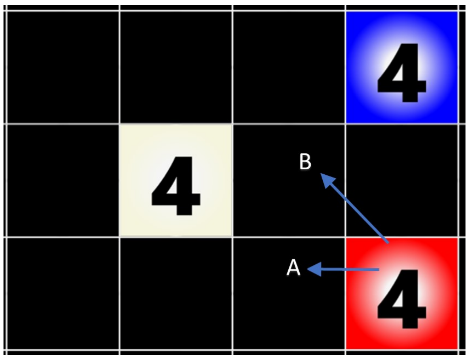


Figure 4: Distance Calculation

Moving in the Board

To move a group of creatures in the board there are many steps involved. First, we have to calculate all possible next moves based on our current position. Second, we apply an algorithm to determine which of these next moves yields the best heuristic evaluation function. Finally, we move in such best direction.
To compute all possible next moves we defined the function "possible_nxt_moves_split" which returns an array containing all available next moves considering both a complete move (all creatures) or a partial one (only a part of the creatures). To do that, it uses another function named "possible_nxt_moves", which basically computes all the adjacent cells given a pair of coordinates, and also the function "valid_coordinates" to take into account the dimensions of the board (we cannot exceed the height and the width of the map). Additionally, this function makes sure that when there are many groups of monsters of the same kind, it is possible to keep some of them in the same position. Thus, complying with rule 1.
On the other hand, to actually move a group of monsters we defined another function, "move_ai", which takes in the source and target coordinates from all monsters of a player and execute the MOV command as many times as necessary in order to move all desired groups.

End of the Game

We also need to define the conditions to determine if the game has finished. To do that we implemented the function "is_terminal". It simply returns true when we have only one kind of creatures in the map or when we have all monsters from both species in the same cell. Otherwise, it returns false.

Algorithm

To be able to make the correct decision at each turn, we choose to analyze the behavior of the game at a certain depth. To find the best move, we create a tree with all the possibilities of game states and find the best game state that works for our AI. However, since this game opposes two players and the opponent's moves are unknown, we must implement an algorithm to be able to predict the opponent's move and respond with the best move. To this end we chose to implement the minimax algorithm.
The minimax algorithm assumes that each player will play optimally and chooses the response accordingly. As a result, the value assigned to the game state at a certain depth is maximized if it is our AI's turn to play, but it is minimized when it is the opponent's turn. The value of the game state that is optimal for the player who is playing is pushed up the tree. Once the value is returned to the root of the tree, we choose the move for the next turn that will maximize our chances of winning.
Using alpha beta pruning is a method to accelerate the search of the tree by stopping the search in certain parts of the tree when we know that the value returned will be higher than a value that has already been pushed up to a smaller depth if the layer above will be played by the opponent and vice versa if the player above is our AI.
We implemented this algorithm in the function "minimax_alphabeta". It takes in as parameters the game board, the current depth, a variable named "is_max_turn" to determine whether it is our turn (True) or the opponent's (False), alpha and beta. The idea is to make a first call of this function in the main program with the correct initial values for all variables:

$$\text{current\_depth} : 0$$

$$\text{board} : \text{game board at the beginning of our turn}$$

$$\text{is\_max\_turn} : \text{True}$$

$$\text{alpha} : -\infty$$

$$\text{beta} : +\infty$$

After that, we call recursively this function from within the function itself until we either reach the maximum depth or the end of the game. At each new call, we have to update the parameters: increase the depth in one, use the updated board with last changes, negate the "is_max_turn" variable to pass the turn to the opponent, and use the updated alpha and beta values.
The "minimax_alphabeta" function starts by asking if we are in a terminal situation (maximum depth or end of the game). If that is the case, then we call our "eval_function" function which will compute the heuristic for the current board. If not, the function is run entirely. It first computes the next possible moves for the active player. Then it iterates through all possible next moves, updates the board accordingly and calls again the "minimax_alphabeta" function using the updated parameters to change to the other player. To update the board correctly, we followed the rules regarding the fights between two types of monsters or one kind of monster vs humans. At the end of each next possible move in the iteration loop, we have the alpha beta pruning algorithm so that we update our target move output only when we achieve a better heuristic. Better means a greater heuristic for player 1 or a smaller heuristic for player 2. We also update our alpha and beta and apply the pruning (break out of the loop) whenever alpha is greater than or equal to beta.
The algorithm runs recursively until we reach a terminal situation, and at the end it outputs the next moves that our creatures should follow in order to maximize their probability to win. This is stored in the variable "action_target". We also need to save the initial position because it is also needed

when using the MOV command. Therefore, this function also outputs a "action_source" variable that has the initial position of our creatures. Additionally, we output the variable "best_value" in order to see what the best heuristic was at each turn.

Heuristic function

As we have seen in the previous section, our method relies highly on the evaluation of the game state. This evaluation is performed by the heuristic function, which takes the information available in the game and evaluates different criteria based on positions, distances and number of creatures to return a single value that indicates the state of the game. If the value returned is high, our AI has a high chance of winning whereas if this value is low the opponent has a better chance. Generally, a 0-value returned by the heuristic function indicates that the game is tied.
We implemented our heuristic function in the "eval_function" module. Our heuristic is composed of many elements. They are all detailed in this section:

Evaluation All Humans. This heuristic will evaluate all group of humans in the map with respect to their closest monster creatures. Thus, we iterate through all group of humans and calculate the closest monster (player 1) and the closest enemy (player 2) to that particular house of humans. Based on this, we simulate a potential fight between the monster (player 1) and the group of humans and define the gain as the difference between the number of monsters after the fight and before the fight. A positive value means a potential win and a negative one a potential loss. Additionally, we compute the distances player1-human (dist1) and player2-human (dist2) and subtract them (dist1 – dist2). Then, we use this difference to determine a coefficient that can be used as a weight. All this can be better described in the following formulas:

$$\text{gain} = \text{future\_nbr\_monster} - \text{initial\_nbr\_monster}$$

where future_nbr_monster is the number of monsters after applying the probability when fighting with humans, and initial_nbr_monster is the number of monsters before they meet.

$$\text{weight} = \begin{cases} 0.75, & \text{diff\_distance} < 0 \\ 0.2, & \text{diff\_distance} = 0 \\ 0, & \text{diff\_distance} > 0 \end{cases} \tag{1}$$

where $\text{diff\_distance} = \text{closest\_distance\_monster\_to\_human} - \text{closest\_distance\_enemy\_to\_human}$, and $\text{diff\_distance} < 0$ means that our AI is closer to the human group than the enemy.
And the evaluation will be simply defined as the multiplication between the gain and the weight.

$$\text{eval} = \text{weight} \times \text{gain}$$

Note that this evaluation is performed for each group of humans in the map. Therefore, if there are many houses of humans, we need to add the partial results to get the final evaluation. The final value is saved in the variable "evaluation_humans".

Evaluation Monster to Humans. This heuristic will evaluate all our monster groups (player 1) in the map with respect to their closest human group. First, we need to define two arrays, one containing the humans closer to our monsters and the other the humans not closer to our monsters (closer to the enemy). Then, we iterate through all our group of monsters and for each of them compute the closest human house first in the closer human array. If there are no more elements in this array, we then compute the closest human house in the not closer human array. These two arrays have to be updated at the end of each iteration in order to not repeat the same human target for more than one group of monsters. We need to have these two arrays because the closest human to the group of monsters is not necessarily the best target. It depends also on the enemy's closeness to such human. Additionally, if two or more group of humans are equally close, our algorithm starts with the one with the greatest number of humans. Once we have determined the human target for our monster, we compute a gain. To do that, we simulate a potential fight between the monster (player 1) and the group of humans and define the gain as the difference between the number of monsters after the fight and before the fight. A positive value means a potential win and a negative one a potential loss. We also compute the distance to such group of humans and use this value to determine a coefficient that can be used as a weight. All this can be better described in the following formulas:

$$\text{gain} = \text{future\_nbr\_monster} - \text{initial\_nbr\_monster}$$

where future_nbr_monster is the number of monsters after applying the probability when fighting with humans, and initial_nbr_monster is the number of monsters before they meet.

$$\text{weight} = -0.02 \times \text{distance} + 0.22$$

where distance is calculated from each monster to the closest human.
And the evaluation will be simply defined as the multiplication between the gain and the weight.

$$\text{eval} = \text{weight} \times \text{gain}$$

Note that this evaluation is performed for each group of our monsters (player 1) in the map. Therefore, if there are many groups, we need to add the partial results to get the final evaluation. The final value is saved in the variable "eval_monster_to_humans".

Evaluation Enemy. This heuristic will evaluate the monster enemy (player 2) in the map with respect to our monsters (player 1). To achieve that, we iterate through all our group of monsters and for each of them compute their closest group of enemies. Then, we simulate a potential fight between our group of monsters (player 1) and the group of enemy monsters (player 2) and compute the difference between the number of our monsters after the fight and the number of the enemy monsters after the fight (difference future). We also compute the difference before the fight (difference initial). Finally, we subtract both quantities (difference future - difference initial) and obtain a difference coefficient which can be seen as a potential gain in terms of our number of monsters vs the number of monsters of the enemy. Therefore, a positive value means a potential win and a negative one a potential loss. We also compute the distance to such group of enemies and use this value to determine a coefficient that can be used as a weight. All this can be better described in the following formulas:

$$\text{monster\_diff} = \text{diff\_future} - \text{diff\_present}$$

where diff_future is the value after applying the probability of survival, and diff_present is the initial state.

$$\text{weight} = \begin{cases} 0, \text{ dist} = 0 \\ 0.9, \text{ dist} = 1 \\ 0, \text{ dist} > 1 \end{cases} \tag{2}$$

where dist is the distance between a group of monsters (player 1) and the closest enemy.
And the evaluation will be simply defined as the multiplication between the difference coefficient

and the weight.

$$\text{eval} = \text{weight} \times \text{monster\_diff}$$

Note that this evaluation is performed for each group of our monsters (player 1) in the map. Therefore, if there are many groups, we need to add the partial results to get the final evaluation. The final value is saved in the variable "eval_enemy".

Evaluation Splits of Monster. This heuristic will evaluate how close the splits of a monster (player 1) are to each other. We deemed necessary to have a heuristic that would evaluate how close the splits of a kind of monster are in the map because being close means that they can merge together easily when necessary. For example, when an enemy with a greater number of creatures is close or when a human with a greater size is nearby. Therefore, we compute the distance in number of plays between the creatures and get a weight coefficient by using the following formula:

$$\text{weight} = -0.02 \times \text{distance} + 0.22$$

Note that this evaluation is only computed when the monster has split. Otherwise, if there is only one group of our monsters it will be zero.

Evaluation Future Monster Enemy. This heuristic will evaluate how our monsters (player 1) would perform on a potential fight against the enemy as a whole. To do that, we iterate through all our group of monsters and assume that we fight consecutively against the total number of enemies. We accumulate our number of monsters after each fight and update the total number of monsters of the enemy. Finally, we subtract those two quantities (future number monsters − future number enemies) and save this difference as a variable "diff_future_total". The idea of this heuristic is to penalize the fact that when our monsters are split it is easier for the enemy to defeat us. On the other hand, having our monsters joined together can give them more strength to fight the enemy. Therefore, this heuristic helps our monsters figure out when it would be a good time to merge back together after being divided during the game.

$$\text{diff\_future\_total} = \text{number\_monster\_accum\_future} - \text{number\_enemy\_future}$$

Global Evaluation. Finally, we implement our global heuristic using all previous elements, as can be seen in the following formula:

Evaluation $= 1 \times$ Evaluation All Humans $+ 1 \times$ Evaluation Monster to Humans $+$
$1.5 \times$ (total number of our monster $-$ total number of enemy) $+ 0.1 \times$ Evaluation Splits of Monster $+$
$0.01 \times$ Evaluation Future Monster Enemy $+ 0.2 \times$ Evaluation Enemy

Apart from the 5 elements described before, we have an additional one, the difference between the total number of our monsters and the total number of the enemy monsters in the map.
As can be seen, we assigned different coefficients to different elements. This is because there are certain components that are more important than the others. In that sense, we assigned the largest coefficient (1.5) to the total difference in number of monsters between player 1 and player 2 because this is the most determinant factor when it comes to winning the game, that is, to have more creatures than your enemy. Next, we gave a coefficient of 1 to both evaluations regarding the humans in the map. We did this because we consider that the easiest way to increase your species in the map is to eat humans as you only need to be as numerous as they are to win a battle and convert them all. Then, we gave a coefficient of 0.2 to the evaluation regarding the enemy. We gave a much smaller

coefficient to this factor because we wanted that our AI would prioritize the humans fights rather than the enemies' ones. This is because we believe that winning the game is more likely if you focus first on the humans in the map. However, this coefficient allows for our monsters to detect close enemies with a number of creatures small enough in order for us to fight and win. But, again, if that is not the case and if winning a fight depends on an uncertain probability, we rather let our monsters go after humans. Finally, we assigned small coefficients to the evaluations regarding the Splits of Monster (0.1) and the Future Monster Enemy (0.01). Though both important, we considered they should not be the main driving force for our monster's decisions. However, there are some situations in which they can prove to be very useful. For example, when there are no more humans in the map and merging our monsters back together could be a decisive step to win the game.

Main Function

Finally, we put it all together in the main function "main", which makes an instance of the class Game (creates all relevant variables and defines all functions) and then we call the function "play_ai" that has all the logic to play the game.
The "play_ai" function iterates indefinitely. Each iteration represents a turn of the player. There will be two different modes for the game. The first one will implement the mini max algorithm based on the heuristic function defined previously. This mode will end when there are no more humans in the map and when our monsters (player 1) are all merged together into one group. After having reached this situation, we will enter into the second mode which basically tells our AI to go after the closest group of enemies in order to finish the game.

Strengths and Limitations

The strength of our AI resides in the heuristic function. We were able to implement a global heuristic that encompasses many components of various natures. In that sense, we evaluate not only the closest humans to our creatures, but all group of humans in the map. Similarly, we not only evaluate the closest enemy to our monster in terms of the distance and the relative number of creatures, but we simulate at each turn a potential fight between all our creatures and the enemy as a whole. In addition to all that, we take into consideration the distance between our own groups of monsters. As can be seen, the combination of all these distinct elements is what gives solidity and soundness to our AI.

On the other hand, the main limitation of our program is the depth of the minimax algorithm. When we implemented a first version of our logic without the split capability, we were able to use a depth of 5. However, when we added the split capacity, we could only afford a depth of 1 because of the time constraints, maximum 2 seconds per turn. Nevertheless, using a depth of 1 for our AI proved to be a more intelligent agent than using a depth of 5 without the split capability. Another limitation of our AI is the fact that it can only split in maximum 2 groups, both of equal size 2. We put this constraint in order to simplify our logic and not overload the algorithm.