Warzone 2.0: Towards Adversarial Agents in a Territorial Geography Game

By Eli Richmond

1. Introduction

This paper describes the re-creation of the game Warzone using the Unity Engine and the addition of several intelligent agents that can play against one another. In order to properly explain the complexities of the state space and the job of the agents, it is necessary to describe the game itself.

Warzone is an online Risk-like game that pits players and bots against one another in a geographical takeover of a given map. The primary outline of the game is simple. Each player or agent begins with 5 armies. Every round, they can deploy the number of armies they have in any permutation to the territories they control, and then they can make a series of attack moves to gain additional territories. If you send more armies into territory then it has to defend, it falls and becomes your territory. There exist larger functions of the map called regions, which are composed of multiple territories, and by controlling one, you gain additional armies that you can deploy each round. The game at this point becomes a race to control regions and to attack opponents. An agent wins the game if all other plays have been defeated.

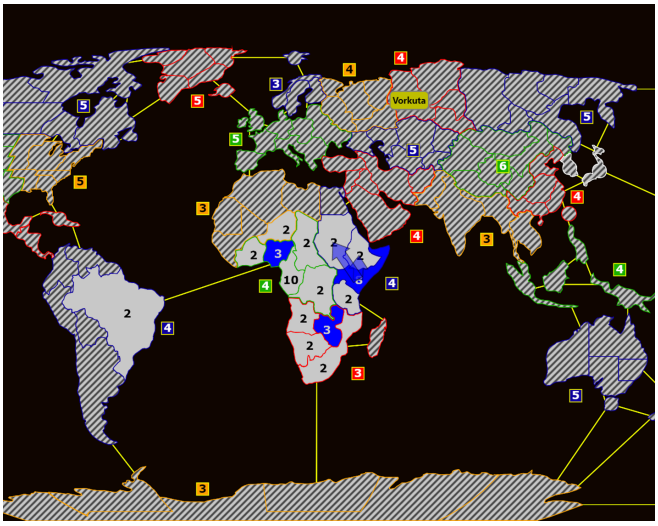


Figure 1. A Screenshot showing a game of Warzone

With the basic premise of the game explained, I went about recreating this game using Unity (to a minimal extent) with solely agents playing against one another, without player interaction. The question I had in mind when designing this project was to assess whether I could create intelligent agents to play this game and if they would exhibit any of the human-like strategies I employ myself.

The premise and intrigue of this game in terms of A.I. is simple: there exists a massive and complex state space wherein agents battle one another and each previous move they make drastically affects their future outcome. It is much like Go or Chess from an agent perspective, except that the state space is technically only limited by the size of the map, and there can be any number of players. In addition, there exist many possible strategies for the agents to employ, including regional prioritization, enemy disruption, and frontline fortification.

The primary challenges I came across when creating the game and agents are numerous and will be described in more detail in the approach section of this paper. As one might suspect, the largest issue came in the form of designing the GameState and how agents would interact with this GameState. In addition, the size of the state space is enormous, so careful consideration was given to how agents search through the GameState and request future GameStates to avoid an impossible amount of needed computational power.

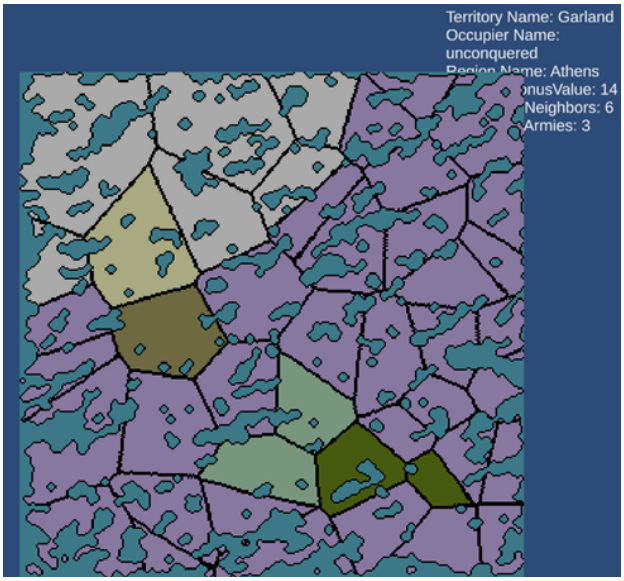


Figure 2. A Screenshot of our re-creation of Warzone

My approach to this problem was relatively straightforward. First, I designed the map generation. Then I designed the visual component. Subsequently, I designed the GameState and how agents interact with it, and then finally I designed and created each one of the agents. I ended up creating multiple unintelligent search agents, such as BFS and DFS, as well as several more intelligent and adversarial agents such as an Alpha Beta agent and a Monte Carlo Tree Search agent. All of these agents were pitted against one another and various factors were measured. It was found that the MCTS agent outperformed the other adversarial agents in a number of factors, including total wins and quickest victories, with the ExpectiMax and AlphaBeta agents performing in a reasonable second place. Comparing these agents however is not very straightforward, and will be described in more detail in the results section.

2. Approach

2.1: GameState and Controller

My approach to implementing non-adversarial and adversarial agents in Warzone begins with the GameState implementation. The monolithic GameState class consists of the map state, which is a list of territories that each have a list of neighbors, with the entire map essentially becoming a large connected graph. In addition, the GameState class provides agents with a way to interact with the game through two types of moves. The first move, called a Deploy Move, allows agents to deploy a number of armies to a territory they have already conquered. The GameState is responsible for verifying the validity of the move, and if valid, will update the number of armies in that territory. The second type of move called an Attack Move, allows agents to move armies around the map from one territory to another. While internal troop transfers can be accomplished

with this move, often these moves involve one agent attacking and gaining new territories. To be clear, you can only attack neighboring territories and you can only use the armies in the attacking territory to attack the neighbors. No troop teleportation or airlifts exist. The GameState is responsible for checking the validity of these moves as well. These two types of moves are often paired with one another, as you can't attack a territory if you don't have any armies to attack with. Later this will be referred to as a Deploy-Attack-Tuple. Lastly, the GameState offers another critical feature: the ability to simulate and play out future GameStates. This is accomplished through a nested inner class that the agents can manipulate that doesn't affect the actual GameState. The last logistic thing to mention is the controller, which spawns the agents and delegates how the rounds progress. Rounds can be visually simulated by clicking the spacebar or can be played out thousands at a time using the data collection feature.

The primary difficulty from the GameState perspective was two-fold. How can I simulate games in a computationally efficient manner, and how can I make sure agents are making valid moves that are being correctly played in the GameState? The first question was answered by allowing agents to create fake GameStates, which were slimmed-down versions of the original class, with territories now decoupled as just a list of strings, and various deep-cloned versions of the rest of the GameState. While this solution functioned, the sheer complexity of the state space was still a limiting factor (as is described below). In addition, a good portion of the GameState class infrastructure was dedicated to validating incoming moves and making sure agents only received valid legal moves to work with. One issue that arose was settling disputes between agents that had both attempted to attack the same territory during a given round. A randomized selection policy was used to determine a victor.

2.2: Naive, Testing, and Search Agents

To begin with, a Naive agent was designed that randomly picks a Deploy-Attack-Tuple legal move each round. As previously mentioned, this tuple exists so the agents can attack with troops they just deployed. This Naive agent works by querying the entire list of legal moves and arranging a random but valid tuple to attack with.

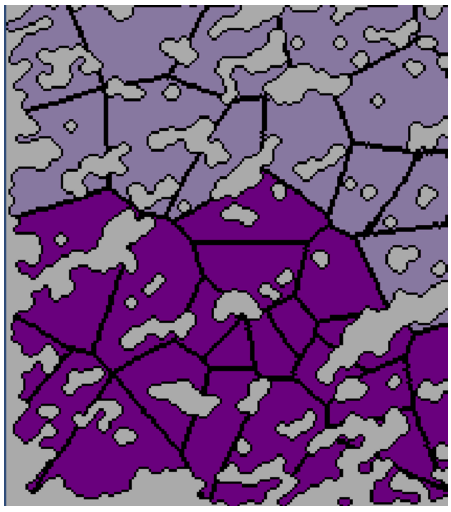


Figure 3. A Screenshot of the Testing Agent playing a game by itself

The Testing Agent is more complex and was designed to push the limits of the GameState and make sure everything was performing as expected. This agent plays like a human and is hardcoded to target regions in order to increase its army count quickly, and then go on the offensive. This agent is unique as it is one of the only agents I gave the ability to deploy/attack multiple territories per round.

Finally, I decided to implement several basic search agents. The first was a non-adversarial DFS agent. This agent would traverse through the graphical representation of the map and would attack in the order DFS visits each node. The BFS agent I implemented was slightly more interesting in that it would attack the frontline in a breadth-like manner, much as its algorithm suggests. Both of these agents were relatively trivial, as this game is inherently meant to be adversarial.

2.3: Minimax, ExpectiMax, and AlphaBeta Agents

Next, I implemented a series of related adversarial search algorithms. The implementation of the Minimax agent served as the base for the other two. To start with, the Minimax agent was implemented to perform a certain depth of Min-max adversarial search each round. This depth was often capped at d=3, as the GameSate grows exponentially with each round, considering the recursive depth increases and the number of possible moves also increases as the rounds progress. Each agent played as its own max agent, with the other agents as the min agents. The algorithm was implemented in the standard recursive fashion, with each min/max agent getting a list of legal moves (in the form of the Deploy-Attack-Tuple). The agent would then generate a subsequent fake GameState which would play the given move, passing it back to the algorithm while increasing the search depth and iterating the opposing agent. The evaluation function for a given move was assessed as the number of armies that move added (if a region was conquered) plus the number of territories gained/lost. This biases the agent to target both regions and territorial gain. I also tried different scoring metrics such as how other agents were hurt by a given move.

```
List<(DeployMoves, AttackMoves)> legalMoves = gameState.generateLegalMoves(agentsList[agentIdx].agentName);

foreach ((DeployMoves, AttackMoves) move in legalMoves)
{
    GameState.AbstractAgentGameState.AgentGameState succGameState =
        gameState.generateSuccessorGameState(move.Item1, move.Item2, agentsList[agentIdx].agentName);
    exploredGameStates += 1;

    (int, DeployMoves, AttackMoves) actionPair = miniMaxRecursive(succGameState, newDepth, childIdx, move.Item1, move.Item2);

    if (actionPair.Item1 > currentMax)
    {
        currentMax = actionPair.Item1;
        maxAction = move;
    }
}
return (currentMax, maxAction.Item1, maxAction.Item2);
```

Figure 4. A section of the Minimax algorithm that was implemented

After a certain search depth is reached, the algorithm terminates and returns the max Deploy-Attack-Tuple it found. This move is then passed back to the actual GameState where it is executed on the map.

The obvious issue that arises here is the state space complexity. At the beginning of a game, assuming we are limiting the search depth to 3, it is not super costly. Each agent has 1 territory and is likely only surrounded by 1 to 5 other territories. Because of the way I structured the legal moves that the agents can query (as the Deploy-Attack tuple), they only have 1 to 5 initial possibilities. As you iterate through the search, the number of possibilities increases on the order of O(b^d) where b is the number of nodes and d is the depth. This is already quite costly, but the true issue exists as you progress through the game. For larger map sizes, as you take more territory, the number of legal moves you have each round increases greatly. If an agent has a frontline of 20 or 30 territories, they have hundreds of potential legal moves per round (and that is for a smaller map). This made it so agents generated tens of thousands of additional GameState classes each round, greatly slowing down the speed of the game. In order to combat this, the next logical step was to implement the Alpha-Beta Pruning version of the MiniMax agent.

Alpha-Beta Pruning performs the min-max adversarial search but restricts the set of possible solutions (Deploy-Atack-Tuples) in accordance with the part of the search tree that has already been examined. In more detail, we can call Beta to be the minimum upper bound of possible solutions, and Alpha to be the maximum lower bound of possible solutions. Before a new node is added to the search tree, we check if it is worth adding based on the current upper and lower bounds that exist already. For example, assuming all agents play optimally, we can say the min agent will always select the minimum, so if the new search node for that specific chunk of the tree produces a result higher than the minimum, we can conclude that the min agent won't pick it and can prune it from the tree. The whole objective here is to narrow down the search tree size and prune (meaning remove) nodes that are not helpful in generating an optimal solution.

```
if (v == actionPair.Item1)
{
    minAction = move;
}

if (v < alpha)
{
    return (v, move.Item1, move.Item2);
}

beta = math.min( x: beta,  y: v);
```

Figure 5. A section of the Alpha Beta Algorithm

Lastly, using the same basic structure, I implemented the ExpectiMax agent. This agent is similar to the Minimax agent, however, instead of maximizing the highest score, the agent attempts to maximize the greatest expected utility. This is because unlike with Minimax, we are no longer assuming adversaries are acting optimally. Instead, we generate chance nodes that take the average utility of all available nodes below them in the tree, which provides the agents with each node's expected utility.

3.4: Monte Carlo Tree Search Agent

Lastly, I implemented a Monte Carlo Tree Search Agent. This agent uses the famous MCTS algorithm to recurse through a tree of GameStates and find the given move for this round that is most likely to help the agent win the game. The algorithm itself is quite simple, but its implementation in my game was not so easy. The algorithm consists of 4 primary steps. 1) Selection, where the algorithm begins with the root node in the search tree and selects a child node with a maximum win rate using Upper Confidence Bound (UCB). 2) Expansion, where the algorithm expands the search tree. 3) Simulation, where the algorithm selects a child node and randomly simulates a game until a playout state is reached (meaning the game has ended). And finally, 4) backpropagation, where the algorithm updates its beliefs by traversing upwards to the root node while updating the number of visits for each node and the node's win rate (the number of times that node won or lost). Each one of these steps is played in order while time remains, so as time increases, the algorithm can theoretically generate a better solution.

```
private NodeT selection(NodeT rootNode) {
    NodeT node = rootNode;
    while (node.children.Count != 0) {
        node = UCT.selectBestUCTNode(node);
    }
    return node;
}

Frequently called  1 usage  Eli
private void expansion(NodeT node)
{
    List<State> possibleStates = node.state.getAllPossibleStates();
    foreach (var state in possibleStates)
    {
        NodeT newNode = new NodeT(state, node);
        newNode.state.currentAgentName = node.state.getOpponent();
        node.children.Add(newNode);
    }
}
Frequently called  1 usage  Eli
private string simulation(NodeT node)
{
    NodeT tempNode = node;
    State tempState = tempNode.state;
    SimulatedPlayoutStates playoutStatus = tempState.playoutStatus;
    if (playoutStatus == SimulatedPlayoutStates.agentVictory && tempState.victoriousAgent == tempState.getOpponent()
    {
        if (tempNode.parent != null)
        {
            tempNode.parent.state.numWins = int.MinValue;
            return tempState.victoriousAgent;
        }
    }
    while (playoutStatus == SimulatedPlayoutStates.inProgress)
    {
        tempState.toggleAgent();
        tempState.randomPlay();
        playoutStatus = tempState.playoutStatus;
    }
    return tempState.victoriousAgent;
}

Frequently called  1 usage  Eli
private void backPropagation(NodeT nodeToExplore, string agentName)
{
    NodeT tempNode = nodeToExplore;
    while (tempNode != null)
    {
        tempNode.state.visitCount += 1;
        if (tempNode.state.playoutStatus == SimulatedPlayoutStates.outOfMoves)
        {
            tempNode.state.numWins -= 1;
        }
        if (tempNode.state.victoriousAgent == agentName)
        {
            tempNode.state.numWins += 1;
        }
        tempNode = tempNode.parent;
```

Now, with background aside, I implemented MCTS as follows. Pieces of the implementation are inspired by a Tic-Tac-Toe Baeldung piece [3]. Each round, an agent receives an updating GameState and performs MCTS for a given number of iterations, usually around 200. This algorithm is usually meant to function online, meaning it runs while time remains. However, there is no true time cap for the amount of time an agent can take to make a move in our game, so I arbitrarily limited the number of iterations MCTS could perform. Next, I implemented a Node class to represent a node in the search tree, which contains information about the State, parent, and children of this node. I implemented the State class to contain the simulated GameState, the move to get to this state, and a ton of other information such as

the number of wins, the number of visits, the opponents, etc.

Random playouts were implemented in a similar way to our Naive agent, and wIe made our version compatible with an arbitrarily large number of opponents. I also implemented Upper Confidence Bound as a selection policy. Additional specifics can be found in the citations, but this policy helps weigh the fundamental question of exploitation vs. exploration in A.I. [4]. As we visit a node more often in the search tree, the UCB value decreases. This helps the algorithm avoid constantly re-visiting the same node which prevents it from further exploring the search tree. Lastly, no heuristic was applied to guide random playouts, so our playouts could be considered light playouts.

$$\frac{w_i}{n_i} + c\sqrt{\frac{\ln N_i}{n_i}}$$

Figure 7. The UCB Equation

4. Results

To begin the discussion of results, it's best to consider a number of things. Firstly, our game of Warzone is predisposed to stalemates. Situations in which all agents control the same number of armies are common and will result in an endless loop of taking the same territory. Because of this, each game is capped at 100 rounds. The results below all come from the same map (unrandomized for these trials).

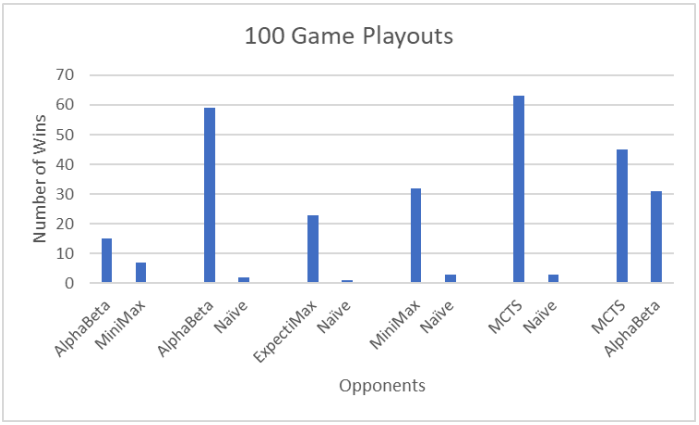
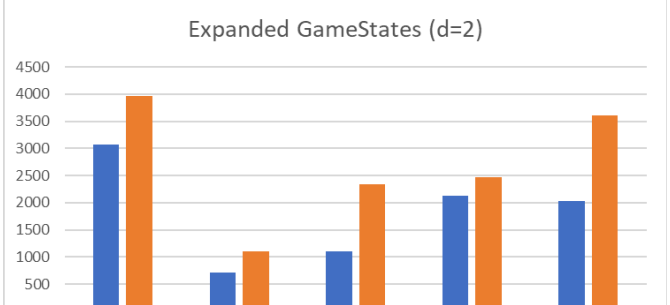


Figure 8. Victories by agent type over 100 games

Firstly, I can discuss victories over 100 games. As should be clear, the Naive agent performs poorly against all adversarial search agents. This is expected as the Naive agent has no metric or heuristic for assessing the potential outcome of a move. Also as expected, in this context all of the Minimax-based adversarial agents perform relatively the same against the Naive agent and one another. The AlphaBeta agent playing the MiniMax agent resulted in almost all stalemates. Interestingly, we can see MCTS had a significant lead over the AlphaBeta agent over 100 games. This is likely the result of MCTS being able to perform more queries in the search tree given the structure of the algorithm, while AlphaBeta was limited due to the time complexity it incurs.



Secondly, it was interesting to see how much of a difference Alpha-Beta pruning made in terms of expanding the search tree. It can be seen that in each one of the games in figure 9, the AlphaBeta agent consistently expanded fewer GameStates, resulting in faster performance.

Thirdly, I also sought to test how quickly an agent could be victorious. As can be seen in figure 10, the AlphaBeta agent took the longest number of rounds on average to win a game when playing against the Naive agent, while MCTS performed the quickest. This is because the MCTS agent tends to prioritize attacking and defeating other agents, rather than gaining territory. This is the case because, during playouts, it is more probable to end the game by attacking an opponent quickly, rather than slowly building up armies. As a side note, the MCTS agent exhibits behavior consistent with an offensive agent in my implementation.

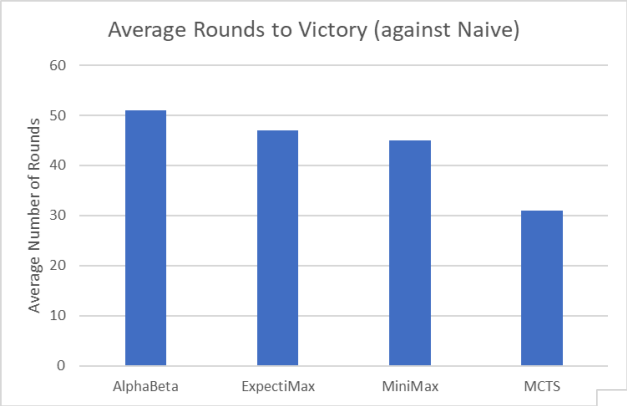


Figure 10. Average number of rounds to agent victory by ag

Finally, I wanted to examine another common characteristic of Warzone, which is taking regional bonuses as quickly as possible. As a reminder, a regional bonus is given in the form of additional armies per round if an agent is able to control a whole region. As I expected, our Testing agent and BFS agents vastly outperform the others, as they are able to take multiple territories per round. However, with all things being equal, we can see the conclusion from the previous graph reflected here. The MCTS agent, because it plays more offensively, doesn't take the regional bonus very quickly, taking on average 20 rounds. All of the Minimax-based adversarial agents take around 18 rounds to accomplish this.

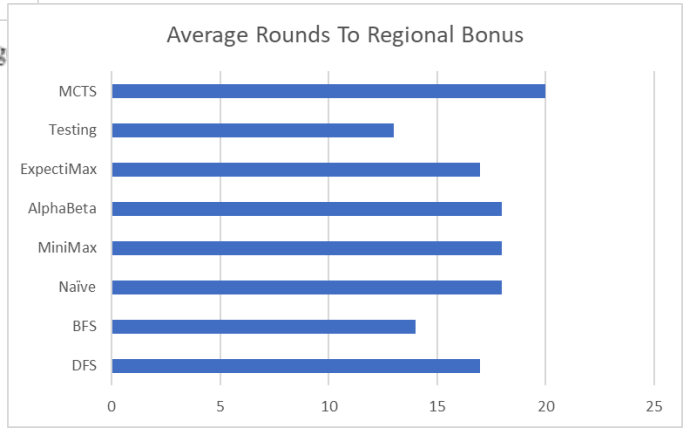


Figure 11. Average rounds to regional bonus by agent

5. Future Work

To reiterate the original purpose posed during this project, "to assess whether I could create intelligent agents to play this game and if they would exhibit any of the human-like strategies I employ myself," it can be concluded that I accomplished this purpose. I implemented several 'intelligent' adversarial agents that could play against one another, and moreover, some of these

agents, especially the MCTS agent, exhibited both an offensive human-like strategy, and the worm-based disruption strategy (knock down an opponent's regional bonus values).
Going forward my goal is to perfect the GameState in a way that will allow for easier expansion of possible GameStates without retaining all information within the current state. MCTS worked extremely well in terms of space conservation and I am interested in implementing a similar algorithm with a re-evaluated set of action values that better represent the impact of each move. This project was born out of my love for the game of Warzone and my curiosity to see if I could implement the algorithms learned in class.

As a final note, the project now has the capability for human players to play against the trained agents.

6. Acknowledgments

The attached code base was created entirely by me, with no external libraries used that are not default to C# or the Unity Engine. The Unity Engine itself was only used to simulate the rendering of the map using Gizmos, but all map generation and pixel positioning was done through our own algorithms.

7. References

[1] Blomqvist, Erik. Stockholm, Sweden, 2020, *Playing the Game of Risk with an AlphaZero Agent*.
[2] Sonesson, Martin. “Creating an AI for Risk Board Game.” *Martin Sonesson*, Martin Sonesson, 7 Jan. 2018, https://martinsonesson.wordpress.com/2018/01/07/creating-an-ai-for-risk-board-game/amp/.
[3] Baeldung, “Monte Carlo Tree Search for Tic-Tac-Toe Game in Java.” *Baeldung*. 3 Octover 2020, https://www.baeldung.com/java-monte-carlo-tree-search
[4] Wang, Benjamin. “Monte Carlo Tree Seach: An Introduction.” *Towards Data Science*. 10 January 2021, https://towardsdatascience.com/monte-carlo-tree-search-an-introduction-503d8c04e168

Figure 1. Ficker, Randy. Warzone. Warzone.com. 2022. warzone.com
Figure 4. Combined Wiki Authors, “Monte Carlo Tree Search.” *Wikepedia*. https://en.wikipedia.org/wiki/Monte_Carlo_tree_search