ECE 368 – Warlight bot Project

Team: Justin Bonner, Paul Chang, Shiva Govindaraju, Milos Malesevic, Ryan McBee, Erik VanderWerf

**Background:**

The bot is made to participate in the online programing competition for Warlight AI Challenge 2 hosted by theaigames.com. The base code for communicating with the API is provided to us by the hosts, and the entire project is coded in C++. As far as the game itself is concerned, Warlight is a Risk-like game, with two opponents fighting for control of a map. The detailed description of the rules can be found on the webpage <http://theaigames.com/competitions/warlight-ai-challenge-2/rules>.

What's specific for this challenge is that map is different for every game, and there is a feature called “fog of war”. Fog of war impacts the game by reducing visibility of the map to the neighboring territories of the regions that you own. This rule had a big impact on the way that we approached the design of our algorithms.

**Overall structure:**

The bot is passive, responding only to the commands from the server. The bot receives information from the API and sends orders regarding the bots actions in the game back to the API when prompted to do so.

At the start of the game, the API takes a region from each super-region and prompts the bot to pick starting regions as initially occupied territories from this list of regions. The bot picks its starting regions – as does the opponent, so all bots start the game with the same number of owned territories.

Each game round, the API prompts the bot with the number of armies the specific bot has available to it. These armies can (ie. should) be placed on any regions the bot currently owns, to increase the number of armies on that region. The bot determines which regions to place armies on, and how many armies to place on it's regions, and returns this information to the API to be updated on the server.

After armies have been placed, the API then prompts the bot to attack/transfer armies from regions the bot owns. The bot determines whether an enemy/neutral region is worth attacking or not, and if yes, how many armies to attack with. The bot will send the API this information in order to prompt an attack. The bot may also decide to transfer armies from one owned region to a neighboring owned region – the prompt for this transfer is done in the same fashion as an attack when given to the API.

Super-regions are groups of adjacent regions that are advantageous to owning completely. Owning a super-region increases the number of armies per turn which a bot can place. Wastelands, on the other hand, are individual neutral regions with more armies than regular neutral territories, making them disadvantageous to conquering.

The game ends when one bot no longer owns any territories on the map.

**Our implementation:**

*Initial Region Picking:*

-Algorithm:

This part of the program selects which regions would be most advantageous to the bot. The server sends a list of regions to pick from, and the algorithm sorts them by a weighting system. It analyses an aspect of each regions' centrality, how many steps away from each region given does it take to enclose the region's super region. The algorithm compares that to the value of the super region of the region. The centrality is subtracted from the value of the super region, and the region with the biggest total is selected.

-Time Complexity:

The entire algorithm completes in O(n) + O(s), where n is the number of regions and s is the size of super region.

*Placing armies*

-Algorithm:

The deployment phase algorithm calculates where to allocate troops for the current round. It does so by assigning weights to every region owned by us, based on the location of the region, and the region's immediate surroundings. These weights are then compared against each other to find the three "highest-need" regions. Armies are then allocated proportionally to each of these highest-need regions based on their weights compared to the weights of the other regions.

-Time Complexity:

O(n \* m), where n is the number of regions and m is the number of neighbors each region has.

*Attack*

-Algorithm:

For the attack section of the program, we examine every region that we own. For every owned region, we generate a vector of the neighboring regions that are suitable for the attack. Firstly, we give the priority to the neighbors that are in the same super-region. If there are none, then we look for the other neighbors. If we still don't find any, we call the transfer function. After that, we examine if we can attack the neighbor with twice the army that he has on that region – if yes we attack with twice his army, otherwise we skip to the next neighbor. In attacking, we give the higher priority to the enemy owned regions, and then we look at the neutral regions. Finally, we return the vector of attacking/transfer moves that need to be placed.

-Time Complexity:

O(r \* Nb), where r is the number of regions that we own and Nb is the number of neighbors of a region.

*Transfer*

-Algorithm:

A modified Dijkstra shortest path algorithm that takes advantage of the specifics of the Warlight map to find the shortest path in a faster time complexity. Because the weight of each edge is one a technique similar to the breadth first traversal of a binary tree can be used on the graph by tracking which regions have been checked. This in conjunction with the fact that the number of neighbors for each region does not increase the the total number of regions in the map increases, allows the path finding algorithm to work in O(n) time.

-Time Complexity:

O(n) for each transfer, where n is the number of regions.

**Team Member Roles:**

Justin: Transfer

Paul: Attack

Shiva: Placing armies

Milos: Attack

Ryan: Placing armies

Erik: Initial region picking