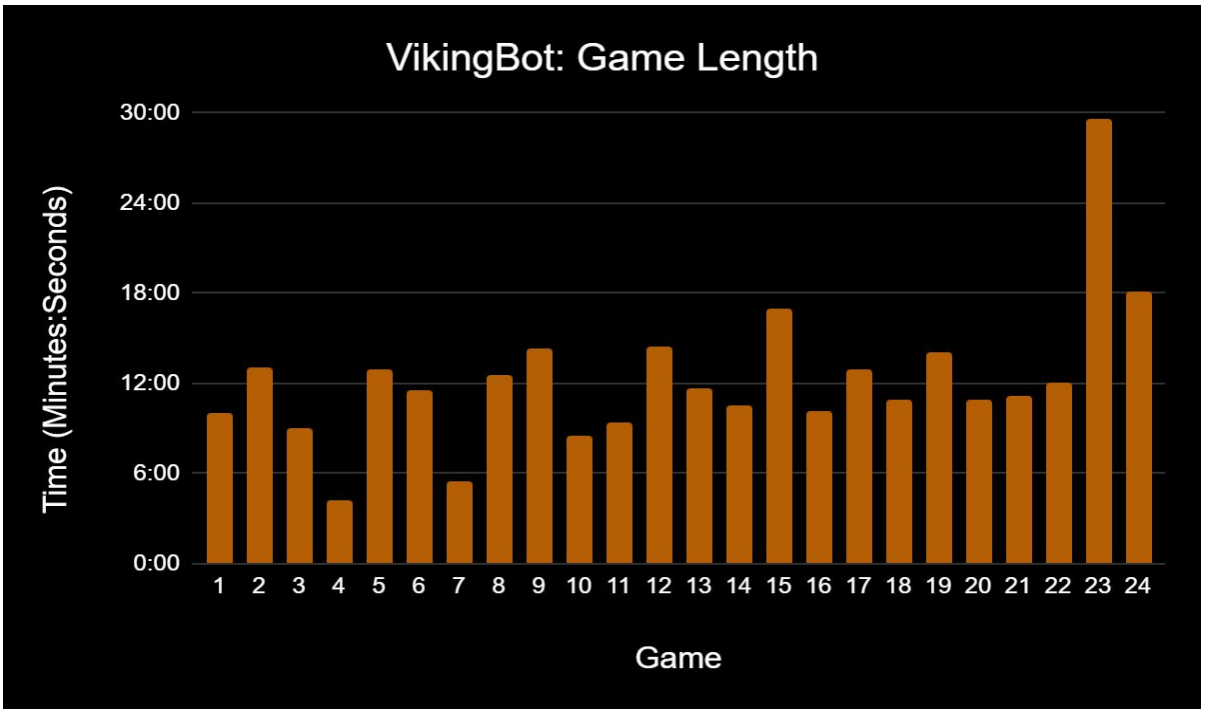
VikingBot

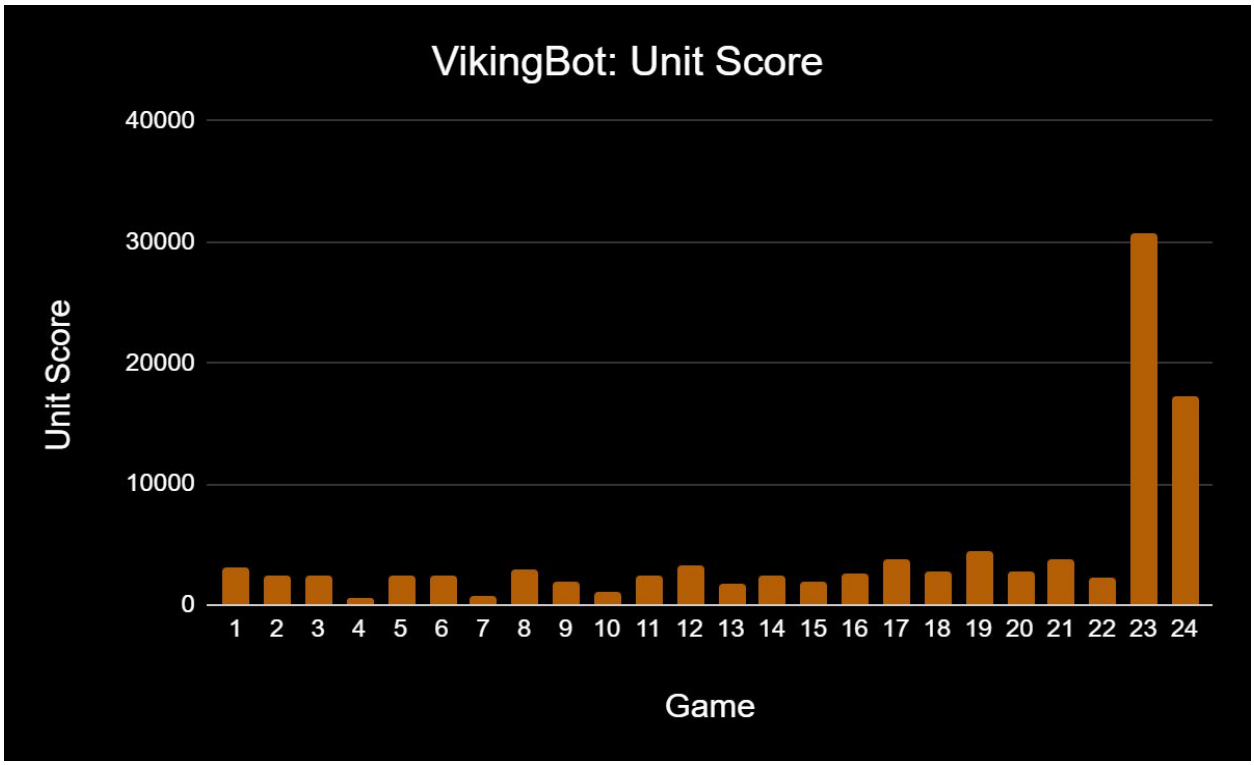
Documentation

Experimental Results

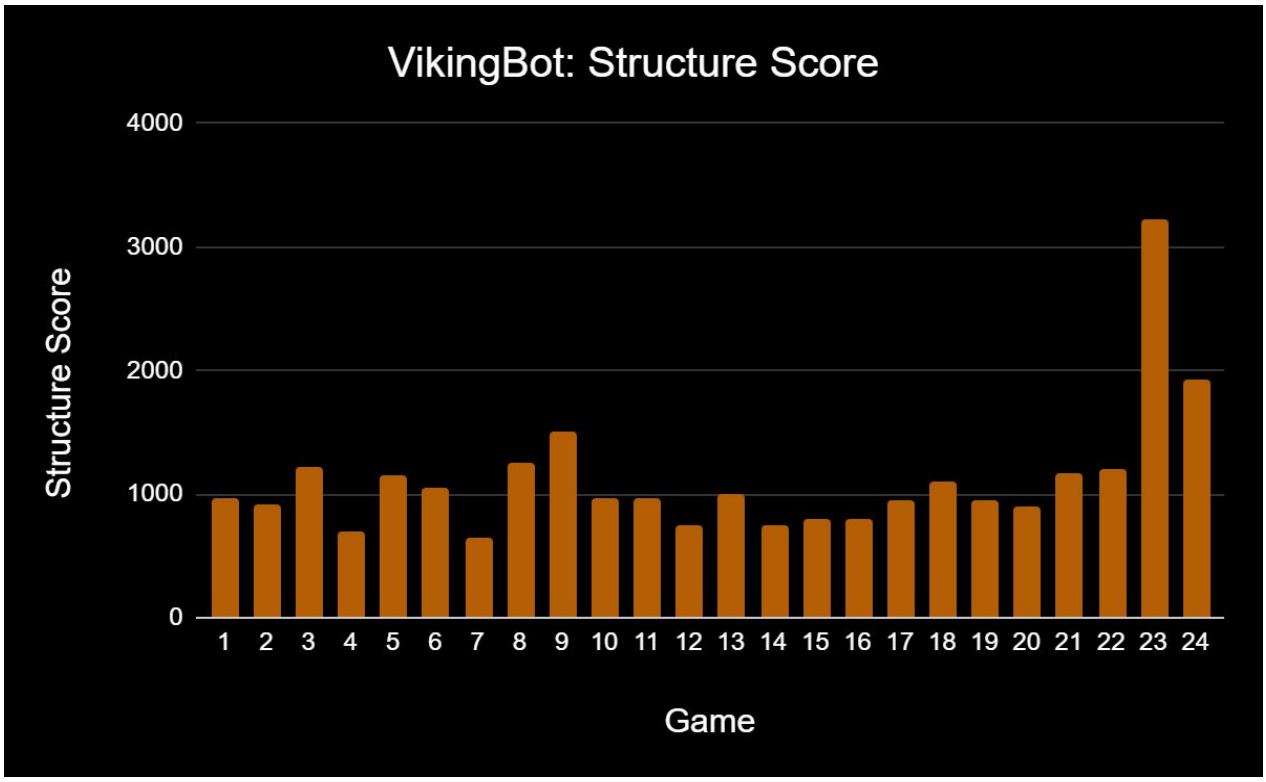
For testing VikingBot played 24 games against the StarCraft AI. These four graphs represent the most important data points to observe.



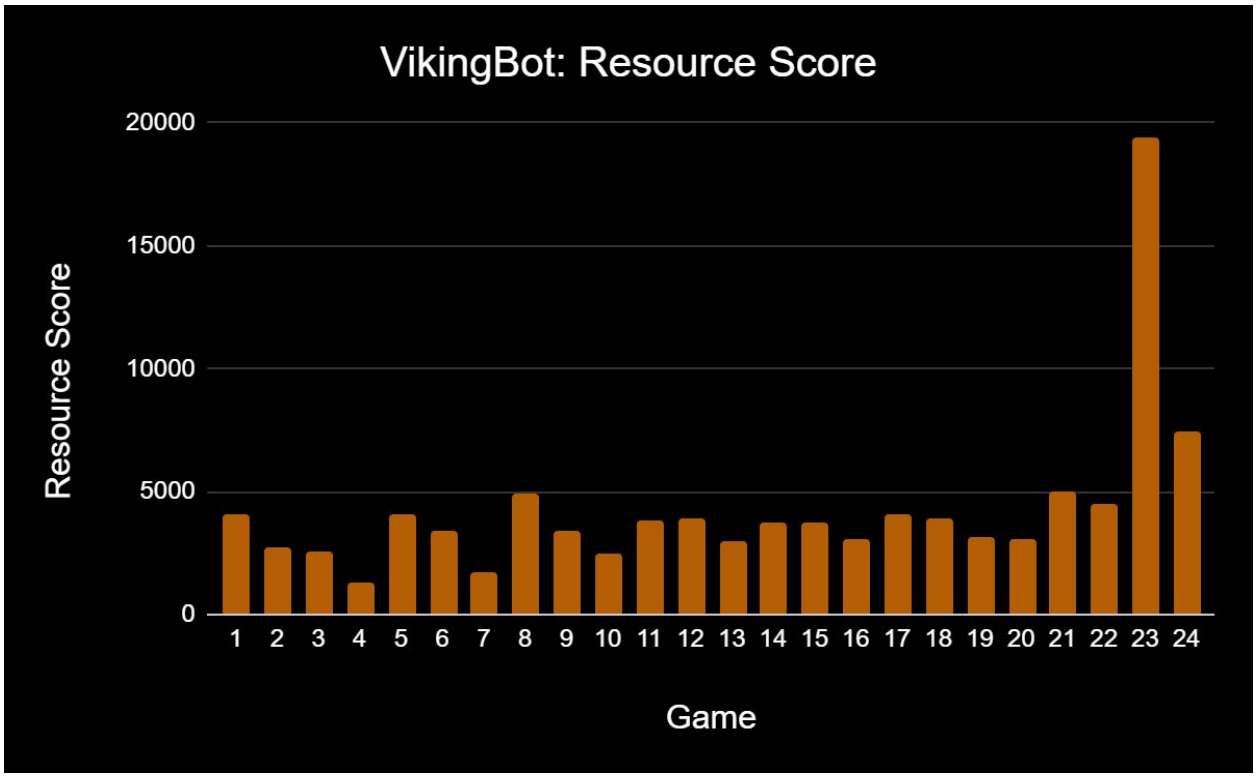
The average game length was 12 minutes 16 seconds. VikingBot only has an early game planning function implemented and typically StarCraft early game only lasts until ~7-12 minutes. Any planning after this would be unreliable as it is still considering the game to be in the early phases. Considering this information, VikingBot performed exceptionally well in game 23 and 24. The shortest games in this dataset were the result of rush strategies by StarCraft's AI, a popular one being Zerg rush. VikingBot does not build Photon Cannons for base defense so these strategies are particularly potent against it.



Unit score is a result of the quantity and type of units constructed or enemy units destroyed during a game. The only units that VikingBot can produce currently are Zealots and Probes. It is no surprise that the longest length games also have the highest scores for all 3 categories. During game 23 VikingBot was able to produce 176 units and destroyed 96 enemy units. To put this in perspective the Average unit score for all 24 games was 4264.58 but without the final 2 games it was 2472.72. The reason that the unit score is so low in most of these games is VikingBot's planning function. Zealot's do not begin large-scale production until certain thresholds are met on Probes and Pylons. When games do not last long enough for these thresholds the unit score is much lower.



Structure score is a combination of the total structures built, enemy structures destroyed, and type of structures built. Game length is also proportional to structure score as longer games have more structures built. Without a Mid-Game or Late-Game planning function in place VikingBot simply spams Pylons when a game passes 20 minutes. Game 23 also saw an expansion with a second base showing up around 16 minutes. Unique behavior was seen after 30 minutes when the planner spam built 8 additional gateways for a total of 12. Although this strategy is not viable in normal games it was beneficial to an extent since the military only consisted of Zealots. By building so many gateways it allowed a strong military force to be created quickly by splitting Zealot production between them. The average structure score across all games was 1121.04 and removing the final 2 games from the equation got an average of 988.86. Structure score is much better overall for VikingBot so the average was not decreased as much when removing the last 2 games.



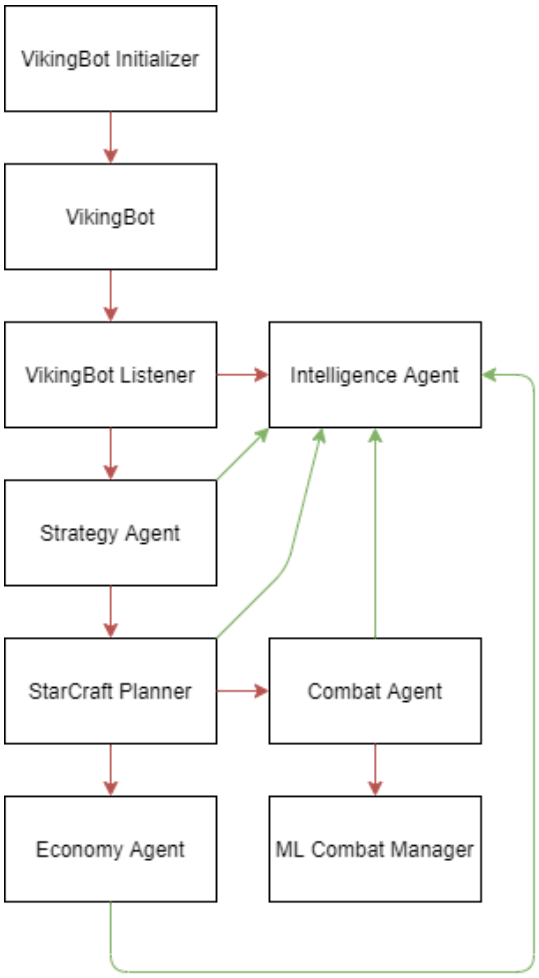
Resource score is simply a combination of gas and minerals gathered. Like the other two scores, the longer a game is, the more resources can be collected. Resource score is VikingBot's most consistent statistic by far because it never stops gathering minerals. Probes will continue to gather resources even when a base is being attacked by the enemy. The average resource score for all games was 4271.5 and removing the final 2 games it was 3438. Game 23 skewed the overall average heavily as it almost quadrupled the previous maximum resource score.

Another relevant piece of data is the win/loss of VikingBot. At the time of this document's creation, VikingBot has yet to achieve a victory against the StarCraft AI. With implementation of a mid-game reward function to create Dragoon units and upgrade unit abilities we believe VikingBot would stand a better chance. The addition of a Forge and Photon Cannon would also counteract the rush strategies that are so effective against it currently. Game 23 was by far the best performance VikingBot has shown and is promising for future work. This game shows that with the addition of the above features and some tuning of the planning function VikingBot has the potential to defeat the StarCraft AI.

Documentation

Summary

VikingBot is a system developed to play the real-time strategy game StarCraft: Brood War. VikingBot utilizes both AI Planning and Reinforcement Machine Learning in order to handle both macromanagement and micromanagement in StarCraft. AI Planning takes care of creating high-level goals that bring the system as a whole closer to winning the game. These goals take the form of attacking, building, expanding, gathering, scouting, training, and upgrading. Then, the goals are then distributed to their respectful agents for execution. A high level view of the structure of the bot can seen in the following image:



Features Implemented

Component	Feature	Description
Strategy Agent	AI Planner	An AI Planning system was created in order to handle the high level macromanagement of the game.
Combat Agent	Machine Learning	Reinforcement Learning is utilized in the micromanagement of combat scenarios.

Features Not Implemented

Component	Feature	Description
VikingBot	Multiple Playable Races	Currently, VikingBot is only capable of playing Protoss and is unable to play as Zerg or Terran.
StrategyAgent	Executing Predefined Strategies	VikingBot doesn't have the functionality for loading and executing a predefined strategy
StrategyAgent	Diverging from Predefined Strategies	Since VikingBot cannot execute a predefined strategy, it also follows that it is unable to adjust a predefined strategy based on the current state of the game.

Maintenance

Requirements

- Java 1.8
- StarCraft: BroodWar patch 1.16.1
- BWAPI / JavaBWAPI
- Chaos Launcher

Initial Setup

- Install StarCraft: Brood War, and update it to version patch 1.16.1, if necessary. Since Blizzard have taken off v1.16.1 and earlier from their website, ICCUP has some

instructions on how to install an older version of BroodWar that is compatible with BWAPI (needs the v1.16.1 patch). Alternatively, there is a BroodWar v1.16.1 hosted here.

- Download SSCAI map pack and extract the included 'sscai' directory into your Starcraft/maps/ folder.
- Get 32bit JRE (Java Runtime Environment) if you don't have it already. Unfortunately, 64bit will not work at this moment. Look for the 'Windows x86 Online' installer.
- Download JavaBWAPI and follow the tutorial for setting up BWAPI.
- Go the the 'BWAPI\Chaoslauncher' folder under Program Files (or wherever you installed BWAPI) and delete the 'BWAPI_PluginInjectord.bwl' file (notice the 'd' at the end of that filename).
- Run the Chaoslauncher ('BWAPI\Chaoslauncher\Chaoslauncher.exe'). We'll always use Chaoslauncher to run StarCraft with BWAPI.
- In Chaoslauncher's 'Plugins' tab, enable 'BWAPI 4.1.2 Injector [RELEASE]' and (optionally, but recommended) 'W-MODE 1.02'.
- Optional: Go to 'Settings' tab in Chaoslauncher and disable the 'Warn about missing admin privileges' option to get rid of some error messages.
- Edit the bwapi.ini file under your 'Starcraft/bwapi-data' folder (with Notepad) and replace this line 'ai = bwapi-data/AI/ExampleAIModule.dll' by this line 'ai = NULL'. All the interesting bot coding related settings can be found in this file.

A full tutorial on how to get started can be found at SSCAI Tutorial which will cover everything from the requirements to getting started with a sample bot.

Dependencies

The VikingBot project utilizes Maven for project dependencies which can be found in the “pom.xml” file in the root of the project. Most modern IDEs (IntelliJ IDEA or Eclipse) should be capable of downloading and configuring the additional dependencies on their own.

Starting VikingBot

The easiest way to compile and run VikingBot is through an IDE such as IntelliJ IDEA or Eclipse. The following steps can be used to run VikingBot through IntelliJ IDEA based on how you would like to run the project.

Direct Run of the Initializer

1. Open the project in IntelliJ IDEA
2. Within the root folder of the project, navigate to VikingBot > src > main > java
3. Right click on the file VikingBotInitializer and click “Run ‘VikingBotIniti….main()’”

Add a Run Configuration

1. Click on the “Run” tab in the toolbar at the top of the window
2. Select “Edit configurations…”

3. Click the plus sign (+) in the top left corner of the new window
4. Select “Application”
5. Give the configuration a memorable name
6. Click the three dots (...) button on the far right of the row that starts with “Main class:”
7. Select “VikingBotInitializer”
8. Click “Ok”

Now you can run VikingBot by clicking the green run button (►) or by pressing the key combination “Alt+Shift+F10”

Stopping VikingBot

There are two ways to stop VikingBot which will be outlined in the following subsections.

Normal Stop

VikingBot will automatically stop on its own once the instance of StarCraft it was connected to exits. At this point, VikingBot will save all of the new data it has learned and terminate.

Force Stop

VikingBot can be forcefully terminated at any point which just involves killing the run through the IDE by pressing the stop button (■). This will cause the run to terminate, VikingBot will not properly shut down/save, and will be disconnected from any instance of StarCraft currently running.

Training VikingBot’s Combat Agent

Create a Training Run Configuration

The training listener is a way to run a minimal form of the bot which only uses the combat agent in order to train the machine learning. This can be done in the following steps:

1. Repeat steps 1 - 7 from Add a Run Configuration
2. Add the string “Training” (without quotes) to the row labeled “Program arguments” within the configuration
3. Click “Ok”

Now you can run the machine learning training by selecting the newly created configuration from the configurations drop down.

Locating the Learned Data from Training

After VikingBot performs a normal termination, it will write its learned data to disk in the directory TrainingFiles > Tables. This directory holds “.ser” files which contain all of the data that VikingBot has learned and stored for melee units and ranged units.

Making the Bot Play Itself

The chaos launcher comes with an exe that allows for multiple instances of starcraft to be launched, the Chaos Launcher - Multiinstance.exe. The chaos launcher can be configured to run an exe file of the bot automatically rather than through intellij as described above. This is required to get the bot to play itself unless you are running two instances of intellij or some other editor.

1. in intellij, compile the bot into a jar.
    a. See this link: https://www.jetbrains.com/help/idea/compiling-applications.html#package_into_jar
        i. Main Menu -> File -> Project Structure -> Artifacts
        ii. Click +, select JAR, then click “From modules with dependencies.”
        iii. Select the main class, VikingBotInitializer
        iv. Hit Apply
        v. Close the menu, and select Build
        vi. Click “Build Artifacts”
        vii. Select the artifact you just created and build it
2. Convert the jar to a .exe
    a. https://www.genuinecoder.com/convert-java-jar-to-exe/
        i. Download Launch4J from a trusted source, like http://launch4j.sourceforge.net/
        ii. Load the lar you just created
        iii. Set the output file to something you will recognize
        iv. In the Header tab, change from GUI to Console
        v. In JRE set Min JRE version to 1.8
        vi. In Version Info, fill out the fields basically however you want to.
        vii. Save the configuration by clicking the Floppy Disk icon to speed up future iterations
        viii. Click the gear to compile the .exe file
3. Move the .exe file to a place bwapi can find it easily (so basically into the bapi data folder)
    a. NOTE: you will also want to copy the ML training data there too so the bot doesn’t start out with dumb combat units
4. set up the bwapi.ini config to work for multiple ai
    a. http://www.starcraftai.com/wiki/Multiple_instances_of_StarCraft
    b. Change the ai = line to include a path to where you put the .exe file
        i. You can have multiple entries to different exe files if you want to test out different versions of the bot against itself, or just one.
5. run the Chaoslauncher - MultiInstance.exe rather than normal chaos launcher, hit start twice
6. multiplayer -> local pc,

    a. Create the game on one instance
    b. Join the game on the other
7. You did it! Start the match.

Menu Automation

The bwapi.ini can speed up testing a lot. First, before anything else, to make this work, you want to change “auto_menu = OFF” to “auto_menu = ON”. This will cause the bot to auto menu to the start game screen. If you’re running a lot of tests, you should change auto_restart to On. the rest of the config file has other options for the game setup. Finally, setting the speed_override would be a good idea so that you’re not waiting as long.

Bugs and Errors

Component	Bug	Description
AI Planner	Gather Gas Spam	If a probe that is currently gathering gas gets stuck or frozen somewhere the Planner will continually spam gather gas actions. This will cause the game speed to slow down to a crawl and use as much processing power as a computer will allow.

Left to be Done

Component	Task	Description
VikingBot	Add Multiple Playable Races	Add functionality for VikingBot to be able to play all races (Protoss, Terran, and Zerg).
AI Planner	Mid-game reward function	Add a reward function to change the bot’s mid-game priorities.
AI Planner	Late-game reward function	Add a reward function to change the bot’s Late-game priorities.
AI Planner	Upgrade Action	Currently there is an upgrade function that does nothing. This should be made to upgrade units.
StrategyAgent	Add Loading of Predefined Strategies	Add functionality for VikingBot for loading a predefined strategy from some configuration file such as json.

StrategyAgent	Add Execution of Predefined Strategies	Add functionality for VikingBot to follow a loaded predefined strategy which tells it what actions should be done in a sequential manner.
StrategyAgent	Allow for Diverging from Predefined Strategies	Add functionality for VikingBot to diverge from a predefined strategy should it find a more optimal route to the end goal.
StrategyAgent	Add Storing of Strategies	Add functionality for VikingBot to write its strategies to disk which will allow for the reuse of strategies and interpretation of generated strategies.
CombatAgent	Add Support for Other Unit Types	Add functionality for the CombatAgent to control other unit types such as air troops.
CombatAgent	Add Unit Clustering	Add functionality to have the ML model learn when to have units cluster together.
CombatAgent	Add Unit Spreading	Add functionality to have the ML model learn when to have units spread apart from each other.
CombatAgent	Finetune Unit Control	Update the CombatAgent and ML model to finetune the control over individual units.
CombatAgent	Add Sequence Rewarding	Update the machine learning model to reward not just a single (State, Action) pair but a sequence of (State, Action) transitions. This will help VikingBot diverge from a greedy action chooser to allow it to gain higher long-term rewards.
Combat Agent	Nuclear Launch Avoidance	Currently, if a terran player tried to nuke the bot, the bot would not respond. There's a function from BWAPI that is called when one happens that could be useful.

Recommended Changes/Additions

Component	**Change**	**Description**
AI Planner	Learned Model	Currently the bot models starcraft through a model that was written ourself. The model is complicated, adds some extra randomness,

		and is relatively slow compared to a Learned Model. Burlap provides the ability to create a Learned Model with some pre-provided implementations. To effectively use this, games would have to be saved, likely using burlap's Episode concept to record and load up games. Additionally, the states would have to be serializable so that they could be saved. The benefit of this is that consequences of an action would only require a table lookup, and the model wouldn't need to be maintained. One other challenge would be recording the results of an action since each action is dequeued at a different time from it's planning, and the result isn't registered in starcraft for at least a frame.
AI Planner	Universal Unit Interpreter	Basically, some way to go from "worker" to scv, or probe, or drone depending on the race being played. Currently these strings are hard coded for protoss, so moving that to an interpreter class that can convert from a string to a unit type given a race would be good for code reuse
AI Planner	Change to Object Oriented States	http://burlap.cs.brown.edu/faq.html#oomdp Right now, our states are just hashable normal states that act as a string based dictionary of objects Benefit: if the current model is kept, rather than constructing a new state every time, an Object Oriented State can just modify individual objects much more easily. Additionally, this means that modifying the fields in the state is easier. Because you don't have to update every single place a new state is created.
Strategy Agent	Chat	If trying to avoid bot detection, sending chat messages might be a good quick addition.

Technical Startup

VikingBot has been thoroughly documented throughout its development using JavaDocs. The generated JavaDocs will greatly aid in the navigation of the project, learning the structure, and understanding its components. The following subsections will outline fundamental knowledge necessary to understand VikingBot, and resources for more information.

StarCraft

- SSCAIT
    - There's a discord server that has some help channels and is dedicated to the SSCAIT
    - The website is old, but has some performance requirements for the bot to be able to compete. As of June 2020, the bot would likely be disqualified due to not being fast enough, in addition to losing.
- Java BWAPI
    - Units are retrieved in a relatively stable order. So, the same worker will be selected for a task before other workers, it seems.
    - Changes can't be detected until the next frame
        - If you give a unit orders, BWAPI will not know that until the next frame when they get sent to starcraft. => you do not know if a unit is free for sure after you give one order. This is not normally a problem though because we only execute 1 action per frame.
        - If you spend minerals on multiple things in a frame, you will not know your mineral count until the next frame
        - In general, this means that the later in the update function something is called, the more likely it is to not be overwritten.
- StarCraft AI

Machine Learning

- Reinforcement Learning
- SARSA

AI Planning

- BURLAP
    - Burlap has many implementations for stuff you may want or need.
    - It's worth noting, our states all have the same fields, and are based on a single class. It is possible to change what fields a state has

The Life of an Action:

1. The planner chooses to take the action via the policy that's been created from the current observation of the game
2. The action is put in the Shared PriorityQueue instance where it may be moved around. Generally if an action requires no resources it will be executed before any action that requires resources, with building a pylon given priority over all actions that require resources.
3. The action gets to the front of the priority queue.
4. In update() in the strategy manager calls canExecute(action) to check if the bot can currently do the action.
5. When canExecute confirms the action can be done, the strategy agent tells the planner to tell the environment to execute the next action. The planner creates a new policy by observing the game. And the Environment tells the appropriate agent what to do.
6. Finally, an environment outcome is generated and ignored. NOTE: the environment outcome would not be accurate anyway because the api calls are unaware of the stuff the bot did this frame until the next one.