Project Report: Live Prediction of Winning a Game and Q-Learning for Building Marines

ECS170, Spring 2018

Group Members (Bot the Builder)

Max Nedorezov, maxned@ucdavis.edu, 913151428
Matthew Quesada, mtquesada@ucdavis.edu, 914953175
Josue Aleman, jaleman@ucdavis.edu 914982971
Ivan Timofeyev, itimofeyev@ucdavis.edu, 998671541
Suhayb Abunijem, sabunijem@ucdavis.edu 91495614

All original members of our team.

Problem and Contribution Statement

In this section, you will describe the problem you want to solve. You should answer the following questions:

- Why is this problem important to solve?
    - Starcraft II is a very difficult game to play using AI due to the huge search space. There are very many possible actions and possible states and training AI to play is difficult because the result of a game only occurs at the end. Using simple machine learning algorithms, it is virtually impossible to reward a certain action because its effect may only be witnessed near the end of a game.
    - Being able to gauge progress of a game may help identify certain patterns that often lead to success, which may aid in training machine learning algorithms to become master players. Being able to provide a heuristic to an algorithm may allow it to learn much faster.
    - Also, while playing a game, there are certain things that are often essential but take a lot of micromanagement such as building marines. A bot that can help micromanage these mundane tasks may make it easier for a player to focus on more strategic tasks.
- Does there already exist a solution to this problem?
    - Blizzard and Deepmind are trying to build AI to play the game without heuristics or prior knowledge. Our solution would oppose this by providing the machine with prior knowledge, which would currently allow a bot to be very good.
    - Even though our solution would use prior knowledge, it may help in the discovery of algorithms that would work without prior knowledge.
    - The built-in scripted AI already plays a game on its own, but we want our bot to teach itself the fastest way to build certain units such as marines.
- Why is this problem technically interesting?
    - It is technically interesting because using our solution, we can get a real time indication of progress independent of who the player is, a real human or AI.
    - Also, the implementation allows us to use real AI techniques to score our own playing as well as our bot's playing.

    - Building a reinforcement learning algorithm will also allow us to discover new techniques/paths of efficiently building certain unit types.
- What possible AI approaches are there to this problem?
    - K-Nearest Neighbors
    - Q-Learning
- What is your solution and contribution to the solution of this problem?
    - Our solution is to look at thousands of replays and classify all of the replays based on whether they are a win or not. We want to find a pattern of units/buildings built in a game that leads to an optimal result, a win. Then, we will be able to score a game's progress based on the number of units built so far and whether that is close to an optimal path.
    - Furthermore, our Q-Learning algorithm will be able to find the optimal path of building marines which will make it possible to have it help a human player in the opening steps of the game to focus more on strategic tasks.
- Why did you choose your solution given the possible alternatives?
    - Finding patterns in large data sets is often difficult but due to the nature of the game, using a classification algorithm will allow us to find a simple pattern much easier. Classifying the number of units built throughout a game will give an easier measure of whether a player is on the right path to win.
    - We will use the Q-Learning algorithm to build marines because they are relatively easy to build using only a few steps. This will make our action and state space much smaller than that of the whole game, but will still allow us to learn practice a machine learning algorithm.

When we initially started this project, we wanted to build a general agent that could execute a general build order and recover from any eventualities that could occur through the course of a game. Such events would have included being attacked and losing a building or technology upgrade that is crucial to execution of the build. In the design stage, we decided that our bot would be primarily focused on macro-management, as opposed to micro-management gameplay. Once the bot completed it's build order, it would perform a simple attack-move command and either win or lose the game based on a single attack.

As we explored more and more options towards how we were going to implement this design, we realized quickly that a vast majority of the reference build orders available on websites like TeamLiquid required every single building and upgrade listed in the builder order, so losing any building or upgrade would require immediate reproduction. In other words, the bot would not actually be doing any strategic design or learning of its own; it would simply be following a pre-defined script and would just backtrack and rebuild anything that was destroyed along the way, every time.

Unfortunately, we did spend a substantial amount of time designing dependency graphs and experimenting with pysc2 before we decided to scrap this idea and move onto another project. We realized that our initial fault-recovery plan was not ambitious enough for the scope of this project, so we moved onto other ideas, even though we had lost a few weeks of precious time already.

In order to recover from this problem and experiment with true AI, we decided to go for a more tangible view of applying artificial intelligence algorithms to starcraft 2, by parsing replays and calculating the player's (or bot's) chance of winning as a game went on, which would help people quantify their chances of wins or losses mathematically as they analyze their gameplay choices and optimize their playstyle. We used K-nearest-neighbors to calculate these probabilities after parsing through a large sample space of replay files.

Additionally, we worked on a Q-learning agent which would try to build marines as efficiently as possible. We had updated our proposal when we were more sure of what we were going to do so at this point, the current proposal reflects our current project. The Q learning agent is able to build marines

although not as efficiently as possible. The technical difficulties and shortcomes of the Q learning agent are further discussed in the appropriate Q learning section of our project on Github.

Design and Technical Approach

- What AI techniques are you proposing?
    - KNN Classifier
    - Q-Learning
- How do the techniques address the problem? Connect the levels of abstraction between problem space and technical solution. Architectural diagrams help.
    - Using KNN, we will be able to classify winning games based on the number of units of each type that are built. Since there are many different types of units, the graph has many dimensions and it would be impossible to classify the replays by hand. By marking games as wins or losses, we will be able to then compare a current game against the classification and figure out if we are on the right path. The classifier will return a probability of a win depending on which path we are on.
    - More specifically, we will have different classifications depending on the race being played which will allow us to compare against the classification since different races require different strategies.
    - For the Q-Learning, we will run many iterations of the game to figure out the optimal way of building a certain number of marines. The algorithm will teach itself the appropriate order of building marines, such as at what point to build barracks.
    - Using the trained Q-table, we will be able to have a human play alongside the AI while it manages building marines while the human is able to focus on more strategic tasks.

- What is your technology stack?

  Development

Unix/Linux
Google style guide
Sublime, Visual Studio Code
Python 3.6
Github

Technology Stack

StarCraft 2
PySC2, StarCraft 2 API

Python
Linux
Google Cloud

- Why is your technology stack the appropriate choice?
    - Python is our first choice for this project because it is versatile and has very readable syntax.
    - Additionally, python has the pysc2 environment and access to Scikit learn, Tkinter, Matplotlib as well as many other packages that will make development fast and easy, giving us more time to work with the gameplay mechanics.
    - PySC2 is already well built for machine learning so it is the appropriate choice for the Q-learner.
- What programming environments are you using?
    - Various text editors and IDEs based on each individual member's preferences (Sublime, Visual Studio Code, Atom, Xcode).
    - Will be running python scripts through the terminal.
- How will you use GitHub?
    - To view all changes to the code we make
    - The master branch will always contain a working version of the AI
    - Branches then pull requests will be created for any changes to the AI as well as different team members separating their tasks from the main branch
- What code quality assurance tools are you using?
    - Github with forks and code reviews before commiting to master
    - We will all follow the Python Google Style Guide

As stated previously, we had initially intended to use dependency graphs for fault detection, but we realized that the end goal was too simple and would not have enough artificial intelligence integrated into our solution. We also had plans to implement many different decision trees that would have been based on the default StarCraft II tech trees but expanded to suit the problem size. The decision trees were also scrapped along with the dependency graphs when we altered our project.

We ultimately decided to take two avenues - one which involved displaying KNN classifier models while executing a replay for gameplay analysis, and another portion which would involve optimizing a marine all-in build using q-learning.

In the original proposal, we stated that we would be using a heuristic to rate the current utility value of a move inside our our state space, and to some extent we still have that technique integrated into the Q-learning portion of our project. The Q-learning agent essentially learns the utility of each action that it takes and observes the result of that action.

If we had the chance to do this project again, we would not have used pysc2 for the q learning bot due to its heavy use of computer vision requirements for sensing the environment. Dealing with pixel values is very difficult and it would have been much easier to use the direct API for that purpose.

Our technology stack mostly stayed the same. We used Python 3 in all of our developing and ran Starcraft II in headless mode in a virtual Linux environment on an Amazon AWS EC2 Ubuntu instance. We continued to create and use our own branches on Github albeit without code reviewing each individual's code.

Timeline

Week 1 (4/30)	~~● Setup environment~~ ~~● Make a bot follow a build order~~ ~~● Research dependency tree techniques~~
Week 2	~~● Make bot smarter by immediately rebuilding buildings as they are destroyed~~ ● ~~Work on the dependency tree technique to make bot even smarte~~r
Week 3	● Parse the thousands of replays provided by PYSC2 for labels we are interested in ● Load the parsed win/lose files to test KNN from Scikit learn (using various k values) ● Build simple GUI to view probability of winning at each time step of a replay ● Apply Q- Learning for simple test case (build 2 marines) and scale up ● Figure out how to train the Q-Learner quickly and efficiently
Week 4	● Interface GUI with live gameplay to view plot of probability ● Apply Q-learning algorithm to help player build 20 marines
Week 5	● Present to class.

When we updated our project proposal a few weeks ago to reflect the new changes we were making to our project, we decided to still keep the first two weeks of work listed in the schedule as a reference to the work that we had attempted to do, but ultimately scrapped.

From that point, we had a few more weeks to implement the new KNN changes and Q-learning solution. The only change to the scope listed on the document at this point is with regards to week 5. Instead of presenting to the class as we initially thought we would. This last week would be spent finalizing the integration of the KNN model outputs into the Tkinter/Matplotlib GUI for validation and analysis, as well as tightening the state space for the Q-learning model in order to further optimize the algorithm as much as possible before concluding the project.

Feasibility

KNN and visualization of KNN outputs

- Planning on implementing a Tkinter GUI with Matplotlib embedded into the application which will display a bot's chances of winning as the game progresses.
- Pre-parse the replay files using a python script which implements the sc2reader API to write out game events into a file. Suhayb has already written such a file which can successfully parse replay files.
- Sc2reader events will be parsed by K-nearest-neighbors algorithm whose outputs can be run alongside the actual replay to visualize the bot's chances of winning as time goes on, in order to quantify good and bad game decisions.

- It is generally easy to get a Tkinter GUI up and running with a small amount of boilerplate code, and there are many resources available for wrapping Matplotlib and other graphing utilities into the interface.
- Matt has experience wrapping Matplotlib into Tkinter GUI's and live-updating datasets, so implementing the live-action graphics should be feasible in the scope of the project.
- The K-nearest algorithm used from Scikit learn has many resources and examples available that allows our group to test various k values and multiple unit type count.

Q-Learning

- Pysc2 is already very well-suited for machine learning applications.
- There are examples of how to build a Q-learner which we will use to better understand the algorithm.
- The hardest part is figuring out how to quickly train the Q-learner with many iterations.
- The state space will be the number of units relevant to the Q-learner such as number of marines, barracks, SCVs, and command centers.
- The action space consists of build marine, SCV, barracks, command center, or do nothing (meaning wait for more mining to happen)
- Very feasible because we have reduced the state and action spaces to be much more manageable for the algorithm requiring much less iteration to converge.
- The Q-Learning algorithm itself is very simple which makes implementation less of a hassle.

Using the KNN algorithm provided by Scikit-learn made it very feasible to quickly fine tune different parameters such as the number of neighbors we want to look at any given time, the algorithm to use such as ball tree, kd tree or simply brute forcing. Our only limitation for this implementation was having to restrict our space to Terran players in order to reduce one less label in the search space. With KNN and the use of the pandas library we have been able to easily slice and manage larger datasets, This project could easily be scaled up to many more replays we'd search through giving us better probabilities for a player to win at a certain time step.

The Q-learner in the end was not as feasible to reach our optimal goal as we had initially hoped. The state space still proved extremely large and training the algorithm proved much more difficult than initially thought. The bot is still simple but it is much harder than expected to have it converge to the optimal result.

Documentation and Access

How are you sharing code? What about a readme? A website is recommended. A Github page or project is recommended.

- We will be using Github and creating forks and making pull requests with the latest changes. The master branch will always have the most recent working version of the AI.
- The Github page will have a README which will be consistently updated as time goes on.
- A PDF of this page will be on the Github repo.

We used Github to store our weekly development logs as well as documentation for each part of the project. All of the information can be found in the following repository,
https://github.com/maxned/Bot-the-Builder

Evaluation

Our implementation will be completely successful if we can have the Q-learning algorithm build a determined amount of marines, assisting the player. We will be able to specify the number of marines we need and measure the algorithm's progress based on how fast it builds the marines using the KNN classifier.

Along with this, the KNN classifier will be successful if we have one of the team members play while a window is able to plot their probability of winning the game based on their current units built in the game so far. If we are unable to do this on the fly, we can save a replay played by one of us, then score it after the game is played.

KNN Classification (Josue)

In using Scikit-learn's KNN classifier we discovered how important it was to have data labeled and indexed correctly. We were able to evaluate over 1k replays which had 91 labels (including the the time step) which contained values such as the a count of Barracks that were present, number of marines, etc. We were not able to successfully interact with a live game and pull data to evaluate with the KNN classifier, as we could not find the right method to have the parser talk directly to the game and then the KNN classifier talk to the parser to create a probability of the current state of the game.

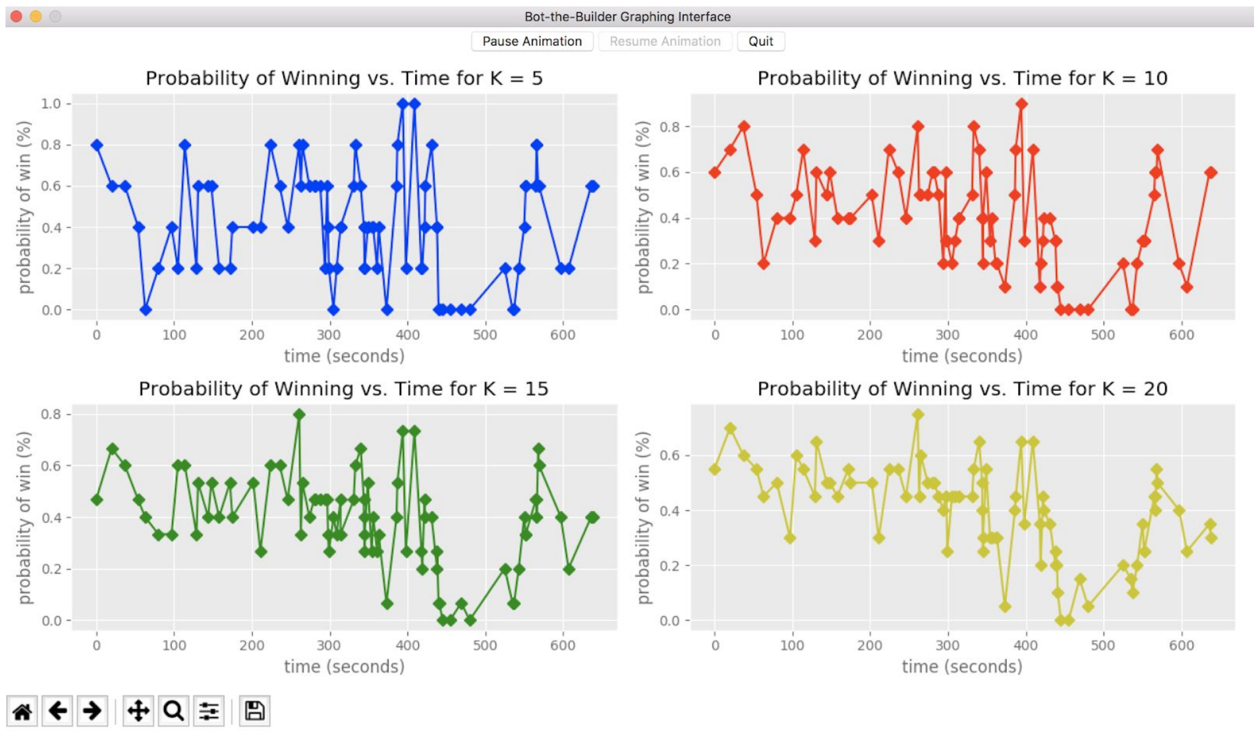
We were able to implement our fallback, which was to save a separate replay and use it to view how the classifier determines the probability the game at each time step. This was successfully implemented for multiple number of neighbors (k). After the classifiers determines probabilities for each game, it is then outputted into a csv file which is used by the KNN GUI to plot. The current setup we have for the values of k is 5, 10,15,20 and we can already see the difference in makes (despite slightly) in determine the probability of winning.

We used a replay in which the player loses and run the parsed replay through the classifier. At 5 which resulted in an average probability of 0.42m, we believe we do not get enough neighbors to make an accurate estimate of winning, but it seemed as though for our dataset, 15 was a good number as the total average of the replay we tested was 0.38. As the number of neighbors increase we begin to make inaccurate decisions as there is more data points being chosen that may lead to some incorrect results.

Overall I believe that having been able to connect live to a person playing would've been of greater value in guiding someone who plays make better game decisions. Another upgrade I would recommend is increasing the number of replays to achieve a much better result.

KNN Visualisation (Matt)

For the analysis and visualisation of K-nearest-neighbors models, we created a Graphical User Interface in Tkinter with contains four live-updating matplotlib graphs, displayed below. The code for this GUI can be found in "KNN Visualisation (Matt)/GUI.py"



The GUI can display KNN models generated by the KNN Classification portion of the project in the form of line graphs. The line graphs display the probability of a win at various snapshots in time, based on all of the information occuring in the game at that timestamp. We ran KNN Classification for 5, 10, 15, and 20 neighbors and displayed all of the results together on the four plots.

Additionally, the GUI has live-updating functionality, and we can continuously load in new KNN data as time goes on and watch the new data render to the plots. The two buttons at the top (Pause Animation and Resume Animation) allow us to temporarily pause live-updating so we can use the matplotlib controls in the bottom left without having the re-rendering process interrupt us.

Using this GUI in conjunction with Suhayb's parser and Josue's classification models, it is possible to quickly analyze StarCraft II replays and view the probability of winning at any timestamp, while simultaneously viewing the replay. This would allow us to see how various player moves in a complicated game like StarCraft II affect a player's probability of winning based on KNN.

Q Learning (Max)

The Q learner was not as successful as initially hoped. The bot was able to learn how to build 20 marines although not very quickly. A lot more information can be found in the README here. Due to the limitations of pysc2, we are not able to play the game with our bot. Pysc2 requires the bot to work with pixel values instead of having direct access to the number of units in the game. This makes it much harder for the bot to talk to specific units and makes programming the bot much harder.

On the other hand, I had problems getting the bot to train to reach the goal of 20 marines. Very often, the bot reached the goal just making random moves because the action space was very small, however, it was not able to converge to an optimum path. All of the things I tried can be found here. The best that my bot was able to achieve was to slowly build 20 marines instead of quickly building them. This was also with hard limits on the number of barracks and supply depots that could be built.

We initially wanted to play the game with our bot and use KNN classification which would have been cool but we were not able to achieve that goal due to not being able to play the game as the bot

builds marines and because it was not possible for us to collect all of the appropriate data using pysc2 during a game. If we had used the normal starcraft API that may have been possible but we chose to stick with pysc2 which was definitely a mistake.

All in all, it would have probably been better to use a genetic algorithm for building marines instead of using Q learning. A genetic algorithm could have been implemented where the gene sequence would encode actions at certain steps of the game. The actions could be the same as in this case. The fitness function would run the different actions at the specified steps and rate each gene based on how many marines it creates and how fast. To reduce the dimensionality of this problem, the generated genes could have a resolution of say 100 steps instead of choosing actions for each step. The hardest part of this algorithm would be generating the genes that encode this information and merging different genes together to make children.

A lot more information about the Q Learning implementation can be found in the README here.

Data collection and parsing (Suhayb)

One of the greatest challenges of data collection is knowing what needs to be collected, and where to source the data. Initially, we were going to use the battle.net API to download thousands of replays. Then we discovered that the replays provided in the research package did not parse well, requiring each replay to rerun the game simulation. The overhead was too high, so we opted to use live data from real game plays. So we sourced as many individual game replays and tournament replays as possible. Then we created a parser that concurrently read 1,311 .SC2Replay replay file to extract critical unit data and exported it into a CSV file. We then read the data from the CSV file and ran a linear regression using TensorFlow in an effort to create a model that can predict the probability of a win. The results of the linear regression were unfavorable with only 61% predictability accuracy which was too close to chance (50%). So, we continued on with our original plan and used KNN to predict the predictability of winning in a particular game state.

Our original goal was to create a bot that can follow and maintain a build order, but after planning out a tech dependency tree we discovered that the problem could be solved by a simple script. The reasoning behind that logic is that units depend on other units thereby if a unit is destroyed the options of whether to proceed with the build order or go back and rebuild the unit is deterministic. In hindsight, with our new found knowledge of logic and planning, we've come to the conclusion that this problem is inherently a planning problem, and should be tackled using planning techniques.

Plan for Deliverables

We plan to showcase it with one of our team members playing live. It will have the help in building for example, 20 marines using the Q-learning algorithm. As the game advances, the live plot will display the probability that the player will win using the KNN classifier.

As discussed in the evaluation section, it was not possible for us to use pysc2 to gather live data of current units and subsequently display the probability of winning a game. Furthermore, our expectations of playing together with a bot using pysc2 were incorrect and we will not be able to showcase that part.

All of the code and documentation for our project can be found on the following Github site,

https://github.com/maxned/Bot-the-Builder

Separation of Tasks for Team

Q-learning implementation to build marines:	Max
Q-learning training setup:	Ivan
Implementation of KNN and packaging output data:	Josué
Using the KNN data, give the probability of a win during a live game:	Matt
Replay parsing for KNN algorithm:	Suhayb

Final Tasks were the following:

Max:	**Implemented, trained, and tweaked the Q learning marine builder**
Ivan:	**Did nothing**
Josue:	**Implemented KNN and packaged output data**
Matt:	**Created a GUI to show the probability of winning a game using the KNN output data**
Suhayb:	**Parsed over 1300 replays and provided the proper format for KNN classification**