Technical University of Crete
School of Electronic and Computer Engineering

Design and Implementation of an Autonomous Agent for the "League of Legends" Game

Dimitrios Trigkakis

Thesis Committee

Associate Professor Michail G. Lagoudakis (ECE)

Assistant Professor Georgios Chalkiadakis (ECE)

Professor Michalis Zervakis (ECE)

Chania, October 2014

Abstract

After the commercial success of the video game "Dota", there has been increasing attention given to the Multiplayer online battle arena (MOBA) subgenre of Real Time Strategy (RTS) games. The creation of agents able to play autonomously within such games is sometimes limited by the absence of a public application programming interface (API). This applies to the popular game "League of Legends", which was greatly inspired by Dota. The few computer-assisted players provided by the designers of this game range from beginner to intermediate, but have direct access to the private API inside the game. This thesis introduces a novel way to handle autonomous agent creation in such games, where access to the game state is limited to the information displayed on the user's screen. The proposed methods come close to what a human player does, since there is a perception phase, which relies mainly on visual analysis, and a decision phase, whose outcome affects the game through emulation of the keyboard and mouse input devices. To achieve this we use screen capture on the game's interface and computer vision algorithms to detect important information. Then, we use artificial intelligence algorithms to encode behaviors for the game character we control. Realizing this perception-decision-action cycle is very demanding in terms of computational resources, however our optimized implementation manages to meet the real-time requirements of the game. Our autonomous agent for the "League of Legends" game is able to achieve intermediate level of play and is quite competent against the designer-provided agents and also against beginner human players.

Acknowledgments

This thesis would be impossible to complete without the positive reinforcement from my advisor M. Lagoudakis, and his constant support. I would like to thank him and my committee for their seemingly endless eagerness to help me with any problem throughout my university studies.

I would also like to thank my family and friends for their involvement in my efforts to keep me motivated and productive during this thesis. You have been great.

Table of Contents

Table of Figures

Chapter 1. Introduction

1.1 Motivation

League of Legends [1] is a video game, where players assume the roles of champions, which they fully control against other human or computer controlled champions. Figure 1 shows a typical screenshot of the game. The environment provides limited resources for which players contest to get a gold advantage. It is thus categorized as a multiplayer online battle arena (MOBA) video game, which is a subgenre of the real time strategy genre [2]. Our goal in this thesis is to create an autonomous agent for the League of Legends game.

Creating an agent for MOBA games is difficult because of the degree of freedom in the player actions. It is widely accepted that a random agent, who selects actions randomly, in a MOBA game cannot even reach mediocre level of play. In games like chess, the available actions are discrete and limited in number, making any choice by the agent seem humanlike, but unsophisticated. In MOBA games, the decision for placement along with the combination of actions available to our champion in real time makes it impossible to make naïve choices seem intentional. Thus, the random agent is almost immediately classified as naïve and inefficient.

Most of the agents made by artificial intelligence (AI) enthusiasts rely on acquiring information about the game state through interfaces that can observe game variables in the memory. However, the information stored in memory is not readily available to human players and some of it is not accessible even through the game's interface. The implementation of AI algorithms is easier and not too costly, when the agent has to just act. But making an agent that has to perceive first and then act, means we get closer to the real goal of AI. This is the case with the League of Legends game, which does not provide an open interface.

Creating an agent who detects the environment through a screenshot of the entire game interface is the other part of the implementation which faces an entire repertoire of problems on its own. League of Legends is a 3d game with 2d interface elements. This makes it easier to abstract the information, since most of it lies in two dimensions and the process of abstraction can be quite reliable. Information which is

natively 3d is hard to acquire and even mapping the 2d abstract map state that we have already acquired from the game minimap, into the 3d space of the main view is challenging. Champions in proximity cannot be reliably detected in two dimensions and we have to rely on their 3d models to know who they are. Rotations and animations increase the complexity of the task. Finally, when we get contradictory information we have to find ways to resolve the dispute.

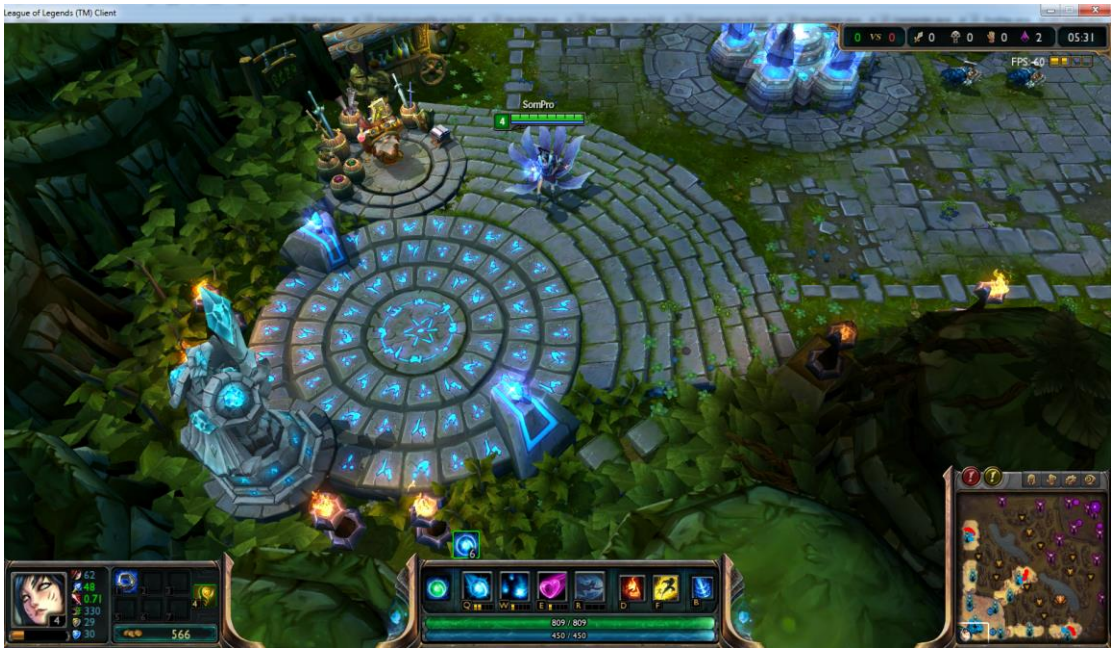

Figure 1. A sceenshot for the "League of Legends" game

For example, there are Bogart problems, which contain two sets of images and there is a distinction to be made between the two sets. The first set could contain small objects, while the second could contain large objects. Problem solving satisfaction aside, they also provided an unsolved problem for artificial intelligence. There had been many efforts to automate Bogart problem solving, but up to a point they relied on humans to abstract the information from the images, before giving them to a computer. The computer would decide what information was missing from one set, but was present in another. Many considered this intervention cheating, since the users that provided the information about the images were thought to put in more information in the system than they got out, making solving the problems trivial. Then, Harris Foundalis made Phaeaco for his PhD thesis, and provided a way to solve the problems by just using the sets of images instead [8]. This work was a great inspiration, because it is in the true heart of AI. Vision is much more complex than any other sense and the primary method for getting information about the world. Adding perception through vision to the agent before he makes

decisions was an important step towards realizing a complete entity that could be considered autonomous.

We wish to implement an agent that can play the game of League of Legends. To do so, he is provided with visual stimuli from the game and tries to abstract visual cues from the acquired information. He then processes that information to derive actions which are deemed to bring the most utility, which are fed to the game using the keyboard and the mouse. By using such a bi-directional interface that is able to snapshot the entire game screen and simulate user input, we create a truly autonomous agent that behaves like a human. To this end, the agent has to process the information of the minimap, which provides the locations and existence of game entities, as well as the view, which is a close-up on the three-dimensional structure of the game. If our agent manages to utilize the tools given to produce more income than his opponents, he will be able to get the advantage in the game consistently. The main objective of the game, destroying the enemy nexus, requires a lot of mini objectives to be completed first. After our agent destroys the enemy nexus, the central building inside the enemy camp, he wins the game.

1.2 Thesis Contribution

The way we tackle our problem involves the following steps. First, we analyze the colors we see in the hsv color space. We use massive parallelization to detect important lines in the view, and after removing the unnecessary ones, we get to detect health bars, getting an indication over where entities are in our view. Then we go to the map, and use hue indications on the locations of buildings to see if they are still enabled. We use the Hough transform method for detecting circles with a specific radius to see where characters lie in our map. Then we use hue histogram vectors to identify the characters. We go back to the view and we use hue histogram vectors to compare the champion in our view to expected champion values to completely identify them, and after that we look at how much health and mana (used for actions) they have. We get these values by letting the character walk over terrain in different orientations and getting mean values. We use a multilayered neural network on the images of digits from our current gold image to update our available gold. We use an imported library for understanding what other players are trying to communicate to us in the chat box. Finally, we make a bag of words system that processes the language.

After we have our game state, we compare the state we got from the map to the one we got from the view. We make appropriate changes to reflect the fact that the view is a bit more reliable than the map, but the map can still give information about things just outside our view.

In the action part of the agent, we form interest points that will guide the character to the most interesting places on the map. A behavioral system based on a vector of "feelings", like "greedy" when minions are nearby or "scared" when enemy turrets are in close proximity to our character, guides which algorithms we should run in the current action phase. The A* algorithm is used when the agent is "scared". For skillshots, which are skills that need to be directed in certain lines, we use a method that minimizes the distance of minions from some line, and then we use that line to shoot as many minions as possible. Finally, we make a complicated scripted system of responses for combinations of "feelings" that will guide the champion to move, attack, use skills, explore, etc. All these responses are based on a coordination-of-actions system that abstracts much functionality and maps them directly to mouse and keyboard actions. The Java robot library provides a way for these actions to be emulated immediately inside the game.

The original goal was twofold:

1. Create an agent better than the random agent in a 1 vs. 0 game.

2. Create an agent that can compete against the in-game beginner bot agents that have access to the private game API.

The solution to the first goal was essentially trivial. As soon as the agent had a basic repertoire of perceptions about the locations of objects, he could easily move around the map and use actions that offer utility. The solution to the second goal was much harder. Since the automated players we played against are at almost beginner human level of play, once we were able to beat them, it showed that our agent did not make naïve mistakes and could be confused for an inexperienced human. However, even small mistakes could lead to very bad performance, so rooting out a lot of sources of misplay was essential. In conclusion, our efforts focused more on making an agent that does not make bad decisions than an agent that makes really good ones. This is why we omitted pure learning approaches and opted for more standard AI approaches that could handle a small repertoire of common situations. Our working hypothesis was that since these situations are very frequent, exceptional knowledge of specific circumstances and invention of solutions to those was not required.

Finally, we were able to test our agent in a fully automated environment. We managed to win reliably against the beginner-level agents implemented already in the game. Interestingly, to a human observer the gameplay of our agent can be hardly distinguished from that of an average human player.

1.3 Thesis Overview

The rest of the thesis is organized as follows. Firstly, in Chapter 2, we cover the background necessary for understanding the employed algorithms. Next, in Chapter 3, we introduce the problem, showing many parts of the game and providing evidence for the difficulty of the problem's multidimensional action space, as well as the probable complications of the visual extraction methods. After that we introduce related work in the field. We notice that there is little work done in the screen capture / action types of agents in virtual environments. In Chapter 4, we describe in detail our approach to the problem, both in regards to visual processing as well as decision making. Then, in Chapter 5, we provide a comparison of the agent to both the beginner-level automated agents implemented inside the game and the beginner human players. Finally, Chapter 6 concludes our work and lists ideas for future extensions.

Chapter 2. Background

2.1 Hsv Color Space

The hsv color space is one of the two most common representations of cylindrical coordinates in color coding. It is used in computer vision because of its closeness to human perception of color. Instead of describing the three components of color as amounts of red, green, and blue (rgb), as shown in Figure 2, we provide three different parameters: hue, which includes all the pure colors without any kind of tint or shade; saturation, which determines how vibrant the color looks; and, value, which corresponds to how far from black the final color is [9] (see Figure 3).

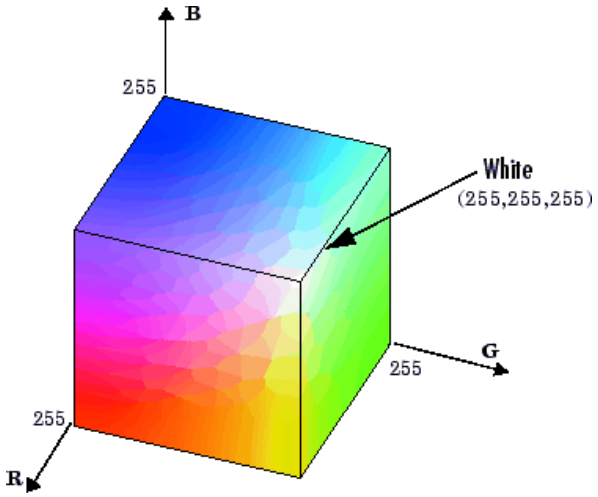

Figure 2. RGB color space [14]

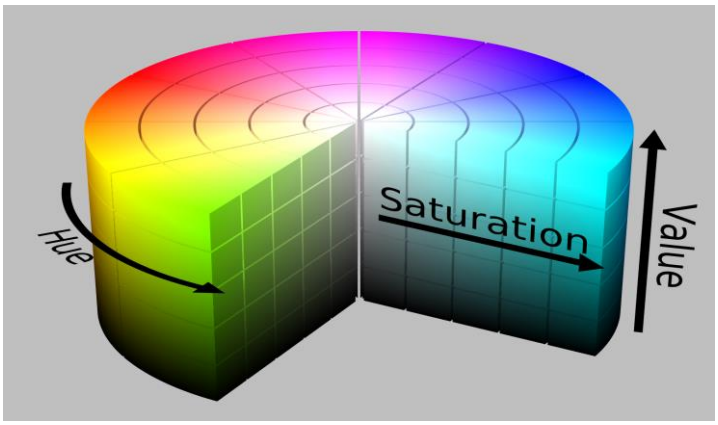

Figure 3. Hsv color space [15]

2.2 Hough Transform

The Hough transform algorithm is an image perception algorithm for the detection of specific features in an image, like lines or circles. Normally, the original algorithm creates an accumulator space, with the parameters of the shapes determining the dimensions of this space. Every point that is detected increases the value of all possible shapes that can produce it in the space [10][11].

Points that are local maxima in our accumulator space represent shapes with parameters determined by the location of the points. We use a disk and circle detection algorithm without using size as a parameter, only location. Thus, we can determine the position of large circles or small disks of a certain color.

2.3 Histograms

A histogram is a representation of the distribution of some data. Color histograms are such representations where the data are the color values of an image [10]. We can make a histogram for every component in the hsv model and acquire information about the distributions of hue, saturation and value inside an image, as shown in Figure 4. To do this, we simply recognize the hsv components of each pixel and count all the pixels that have the same hsv component. We first decide the ranges where the component value of a pixel must fall in to get accounted for by the counting process. Then each range's count is shown in the histogram. The hsv histogram that results from this process contains very important information about the image and can be used for classification.

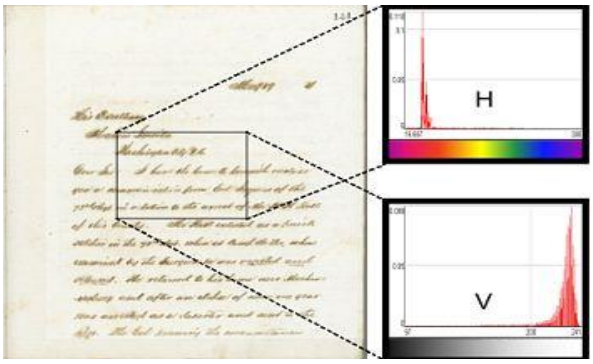

Figure 4. Hsv histogram (showing the Hue and Value components) [17]

2.4 A* Algorithm

One of the most successful search algorithms for pathfinding is the A* algorithm [5][6]. It uses a best-first search strategy to find a path from the starting point to the end point. It is applied on a system of interconnected nodes, namely a graph. If we can traverse between all the connected nodes, with a movement cost provided by the weight on those nodes, we can apply the algorithm to find the shortest path between any two nodes in a very accurate and efficient way.

The algorithm maintains a set of open nodes, the set of nodes we have not evaluated yet, which initially contains only the starting node. For every neighbor node of our currently evaluated node, we calculate the cost to reach it. It is equal to the sum of the past path-cost function g(x) and the expected future path-cost function h(x). Function h(x) heuristically estimates the remaining cost and has to be an admissible heuristic, meaning it should never overestimate the distance between the current node and the target node. After we find the node that has the lowest f(x) = g(x) + h(x) cost, we obtain our next node for evaluation. Every node that has not been evaluated yet remains in the open set. Every node that has been evaluated is put in the closed set. When the end (target) node is popped for evaluation, the algorithm terminates.

The A* algorithm keeps track of each node's predecessor. After the algorithm ends, the ending node will point to its predecessor, and so on, until we reach the starting node and this way we can get the full path.

2.5 Linear Regression

We are given a set of points (x,y) that lie on the plane and we are asked to find what the principal directions are, that is directions in which the set of points varies the most [4]. To do this we used Deming regression.

We start by realizing that the "errors" between the points and our lines have the same scaling in both x and y directions. That is, the vertical and the horizontal axis have the same measure. This leads to a delta value of one. To find the best fit, we first calculate certain quantities, because the solution can be expressed in terms of the second-degree sample moments.

Solving for the three variables, we find the best fit for the line equation $y^* = \beta_0 + \beta_1 x^*$, where we are expected to minimize the weighted sum of squared residuals of the model (see Figure 5).

$$\begin{aligned}
\overline{x} &= \frac{1}{n}\sum x_i, \quad \overline{y} = \frac{1}{n}\sum y_i, \\
s_{xx} &= \tfrac{1}{n-1}\sum (x_i - \overline{x})^2, \\
s_{xy} &= \tfrac{1}{n-1}\sum (x_i - \overline{x})(y_i - \overline{y}), \\
s_{yy} &= \tfrac{1}{n-1}\sum (y_i - \overline{y})^2. \\
\hat{\beta}_1 &= \frac{s_{yy} - \delta s_{xx} + \sqrt{(s_{yy} - \delta s_{xx})^2 + 4\delta s_{xy}^2}}{2s_{xy}}, \\
\hat{\beta}_0 &= \overline{y} - \hat{\beta}_1 \overline{x}, \\
\hat{x}_i^* &= x_i + \frac{\hat{\beta}_1}{\hat{\beta}_1^2 + \delta}(y_i - \hat{\beta}_0 - \hat{\beta}_1 x_i).
\end{aligned}$$

Figure 5. Equations for solving the linear regression problem

2.6 Neural Networks

In machine learning, neural networks are computational models that are used for pattern recognition (see Figure 6). They are represented by a network of interconnected nodes called neurons which can perform computations based on input [7].

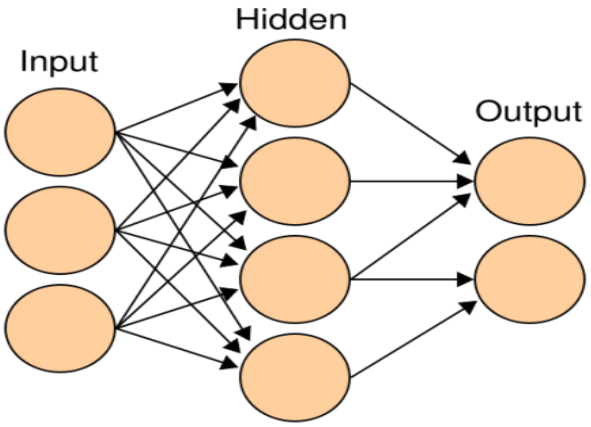

Figure 6. Neural network with three layers of neurons [16]

What has attracted the most interest in neural networks is the ability to learn. In supervised learning, we provide a set of examples (x,y) where x is the input vector and y is the expected response to the input. The network adjusts the weights between the nodes so that it can approximate the function f(x) -> y. Even if the input does not conform to a specific example, the network will reach a solution based on generalizations from the samples based on trained weights. To actually train the network, we need an algorithm that will change the weights according to what the input-output pairs are. In this case, we used the resilient backpropagation algorithm [13].

2.7 Optical Character Recognition

Optical character recognition (OCR) gives us the tools to extract character information from visual images. The algorithm employed is given an image containing a digit as input and has to decide which digit it is [7].

In the training phase, the algorithm uses a set of randomly generated scanlines over the image that either cross or do not cross the digit. If a line crosses about half the digits in the set, the line has a high entropy value and can be used to differentiate between the digits. We keep a small number of lines, the ones with the highest entropy, as features. When the lines cross the digit we input 1 to the corresponding input node of the neural network that we will use to classify the digits. If the line does not cross the digit we input 0 to the corresponding input node.

After training the network for a large number of samples, we get the classified digit in the output layer of the neural network.

Chapter 3. Problem Statement

3.1 The "League of Legends" Game

In League of Legends [1], the player starts the game at one side of the map. Every time he has enough gold, the main currency of the game, to buy items, he may do so whenever he arrives at the main platform of his base. In every game there are two competing teams. These teams are comprised of up to 5 players each, and their bases are located in opposite corners. Players have to defend their nexus, which resides in their base, and destroy the enemy nexus to win the game. At first it is impossible to actually reach the enemy nexus without dying, creating what is known as the lane phase. There are three lanes where champions fight for gold. Little entities called minions arrive at the lane from each side. The players have to kill the minions at the last moment to get the gold otherwise they get nothing. Turrets, buildings made to protect the nexus, attack anything in close proximity. Turrets provide gold when they are destroyed, and they focus on minions if the player is not aggressive towards enemy players beneath them. This provides the main way to win, beating your lane opponent and after that, destroying enough turrets to get to the enemy base. Buildings called inhibitors have to be destroyed before the enemy nexus can be attacked. Unlike turrets, they revive after some time, but when they are destroyed, the ally nexus produces stronger minions. Finally, there are resources for gold between the lanes, in what is called the jungle. The jungle is not used by the agents implemented by the game, so we will not be using it either. However the jungle sometimes provides faster paths to move from one point to the next.

Our agent has to buy items, go to his lane, fight with enemy champions over minion gold, attack minions at the last moment to get that gold and follow his team in team fights. He has to know how much gold he has, where to be approximately so he can travel there when it is important, how to attack enemies by combining actions and where the enemy and ally turrets are (when they are enabled). Another feature which is not used very often is the chat box, where allies can communicate with our agent to instruct him to go to certain places. Finally, he has to know when to recall if he is in low health and when

to look at enemy turrets by allowing the view to not have him at its center.

To do so, he has access to a minimap which contains strategic information about the location of buildings, minion and champions. He also has access to the view, which contains the landscape and the objects that are placed in it in close view. We use both, as well as some interface information to abstract the game state into our own model.

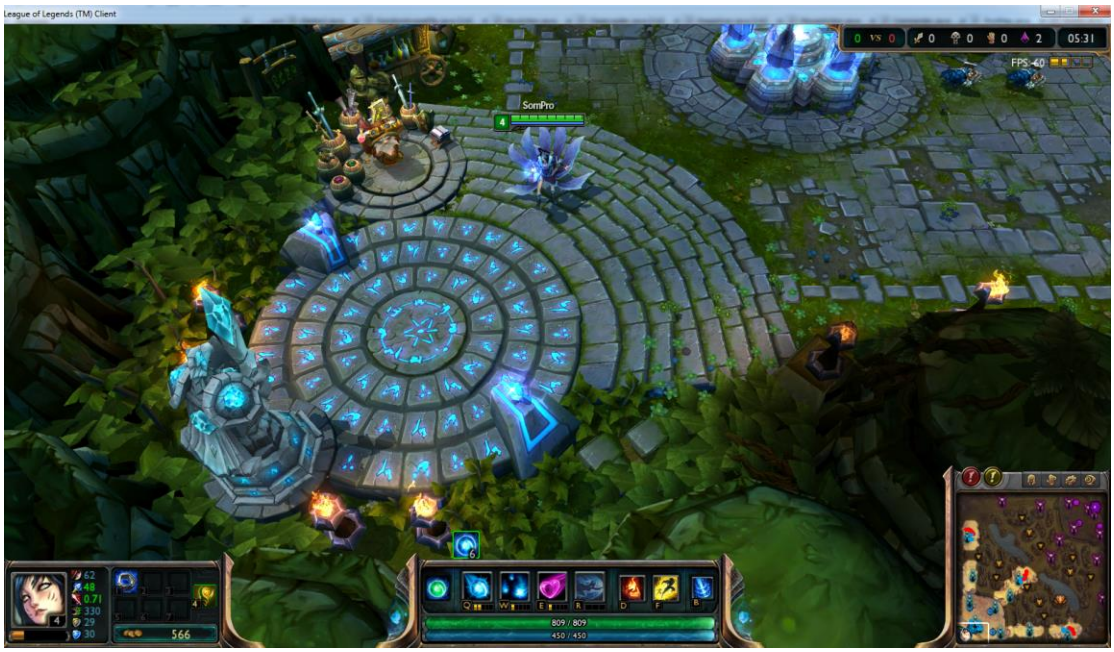

Figure 7. Main view

The main view of the game (see Figure 7) shows features, such as the minimap in the lower right, the skill section in the middle, and the player's items to the left. This is the starting point for all champions after the game begins or after they die and their death counter reaches zero. After that, agents are free to roam around the map and acquire gold. Our character is also visible, and we can see the health bar that is associated with him. The green part shows our character's health, while the blue part shows our character's mana, used for actions.

Figure 8. Character health bar

Figure 8 shows our agent's health bar. We use the lines that separate the health into parts to count how much health our character has. Every vertical black line corresponds to a 100-value increment for our current health.

Figure 9. Non-lane terrain / "Jungle"

Our map processing system provides clues as to which jungle monsters are alive (see Figure 9). Our character does not want to attack these monsters and they are left for the player whose specialty is getting those monsters in the jungle. However, we detect them both in our minimap, to see if they are available for the taking, as well as in our view, in the same way we view normal enemy minions. Since they are not close to dying, our agent will not attack them.

Figure 10. Lane view

This is a view of the lane (see Figure 10). Inside it, we see ally and enemy minions fighting each other. A "wave" of minions grants protection against certain champion skills as well as turrets, which target minions first unless our agent provokes enemy champions beneath them.

The health bars are surrounded by black lines which contain either blue or red filling based on whether the minion is an ally or an enemy of the team (see Figure 11).

Figure 11. Minion health bars

Figure 12. Turret health bar

In this view (see Figure 12), we see that ally turrets have health. The buildings they protect have no health bars until the turrets are destroyed. The nexus is protected until both turrets fall (see Figure 13).

Figure 13. Enemy buildings in "fog of war"

We can also see that enemy buildings are hidden from the fog of war (see Figure 13). We can see whether they are there, but we cannot see how much health they have. This is a problem when we want to see things outside our character's range, since we do not know if the building is covered in fog of war or is simply missing. Since detection is mostly done through health bars, this becomes a hard problem that persists through many feature detection algorithms.

3.2 Thesis Goals

What we are trying to implement runs across two dimensions. First we want to detect the features accurately. This will provide the missing API we need to connect our agent to the game. To complete this connection, we need to make sure the substrate is there to support our actions inside the game. Thus we need to abstract the keyboard/mouse input system, and create an accurate representation of the game state based on the screenshot.

After we do this, we have a small amount of resources, measured in processing time, to actually implement our behavior. Our agent has to obtain more gold than the opponent. To do so, he has to decide where to be and what to do. Positioning is mainly done through the minimap, with clues as to the whereabouts of enemies or potential for gold. Defending turrets against enemy champions and minion waves is also important, because denying the enemy gold is a valid tactic.

We obtain gold mainly through last hitting enemy minions. This means that our perception of the view must be very accurate and detect all the important entities on the screen, as well as record statistics about them. After we know how much health the enemy minions have, we must organize our attacks so that we can last hit them to gain gold. We need to coordinate a lot of actions, from attacking minions, to going back to avoid sources of danger and engaging enemy champions.

Finally, we would need to be able to understand communication from teammates (if not outright respond to them) and recognize the amount of gold we currently have in order to buy items.

If we do all these, our agent will be able to get gold early in the game, buy items and be a force to be reckoned with later in the game, where killing enemy champions and destroying enemy buildings becomes a priority. If the management of our position and gold is sufficient, we will be able to win by destroying most of the enemy buildings (enemy nexus included) and winning the game.

3.3 Related Work

A similar problem was presented at the IEEE Computer Intelligence and Games conference, where a competition was held with the purpose of creating a controller for the game "Ms. Pac Man" [3].

The objective was again twofold, first use the screen capture method to obtain the game state, and afterwards act in the best interest of the agent. The same kind problems were present in this version of screen capturing agents since firstly, non-determinism and secondly, screen capture not accurately reflecting the current game state, since some time has already passed from the moment of the screen capture.

It is arguable that in "League of Legends" the game is deterministic, however unlike "Ms. Pac Man" where the game state is extracted easily, in League of Legends we have non-determinism because of the amount of visual information that cannot be extracted accurately. Overlapping causes serious issues in determining the exact game state, and that along with the delays in action, which can reach important fractions of a second, become a similar source of inefficiency.

The methods employed for screen capture in the game of "Ms. Pac Man" were adequate to capture the entire game state in less than ten milliseconds. We are not granted the same privileges since extraction of an incomplete game state in "League of Legends" requires an amount of milliseconds in the hundreds. Since required response times are similar between the two games, we had to use the time allotted more efficiently and without using too many resources.

Thus, in a way, this problem proves to be much more complex than the "Ms. Pac Man" controller implementation and our agent has to make tradeoffs between efficiency and resource management.

Chapter 4. Our Approach

4.1 Visual Cues, the Map System

Our starting point is the snapshot of the map view (see Figure 14). Initially, we just want to get the rgb value of pixels inside the image and transform it to a more useful form.

Figure 14. Initial Map Screenshot

The transformation from the rgb color space to the hsv color space is essentially a mapping between red, green and blue components to hue, saturation and value (brightness) components.

The rgb color space is convenient because currently available computer monitors produce light of different wavelengths to produce any color on the screen. However, humans classify colors not based on these components but based on the hue which is the pure color category, like yellow, the saturation which shows how faded the color is and value which is how bright it is.

We can clearly see that there are many colors to be used in the recognition of map entities. We need to reduce the amount of information in the map so that the possibility for background noise and variation to alter our results will be insignificant.

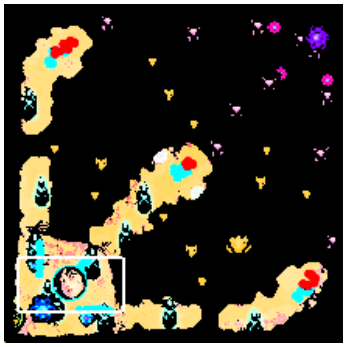

Figure 15. Quantization of the map screenshot

After this initial phase, we quantize the results so that the visual information is clearer (see Figure 15). For every component of the hsv color space, we quantize the range uniformly using a small set of discrete values.

We have already lost some of the available information, for example wall placement, but the tradeoff is that we also gained clarity. It is also clear where our agent has visibility on the map and where there is visual obstruction, also known as "fog of war".

We already see that noise has entered the picture and that there is no exact match for any kind of target image we want to detect. The goal is not simply to match a small target image pixel for pixel, but use a method that is resilient to background noise and variation.

First, we remove the orange parts inside the map, as they correspond to visible areas. We can already see some degradation in the quality of our champion's image, but we can also see that what remains is the information that is important, and processing can begin (see Figure 16).

Figure 16. The map after the removal of visible areas

Since we do not want to lower the quality of our image, after detecting the location of features we use the original, not the quantized, map to detect the exact colors in each one. In features that location is already established, we use scan boxes on the original image to form histograms. The term box is a bit misleading since these mini images take either square or circle shapes to accommodate different kinds of features. They are taken from the original image of the map. The histograms that are formed by these snap boxes are essentially vectors of hue, saturation and value that we are able to use to determine whether the location contains the feature or not.

As already established, histogram vectors are objects that contain the hsv values of scan boxes on the minimap. We first examine predetermined locations of a feature and then create a disk on the image that analyzes the pixels present on it. This process forms a histogram vector, which is a count over all of these accumulated pixels, which can be used as a prototype or blueprint of the original feature. Sometimes the thresholding will be done on part of the vector. If the image contains other features or the original feature we are looking for is missing, the distance from these histograms or parts of histograms becomes large and doesn't meet our thresholding criteria. When these criteria are met, we update the state to reflect the fact that we found the feature we were looking for.

The scan area type is a disk because most features have complex shapes that generally fit inside the circle, meaning that if other pixels outside the disk were taken they would just help to introduce noise and variation in our sampled features, making meaningful comparisons with the prototype vector difficult. These prototypes are sampled in-game and serve as representatives of the features they are taken from. The histogram vectors shown are concatenated, meaning that each component (hue, saturation and value) is represented by one third of the image. The coloring scheme helps understand what each component represents inside the vector. An example of a histogram vector is shown in Figure 17.

Detection of buildings and the jungle

We begin our detection of buildings and jungle monsters by using the histograms generated by the disk shaped snap boxes on preset locations. We select certain ranges inside the histogram, and use them for thresholding. If the count of pixels inside that range surpasses a certain value, we consider it a valid detection of the feature. Buildings come in two varieties. Enemy buildings are mainly purple and allied

buildings are mainly blue. We use a high value in the hue section of the histogram that represents these colors to identify the buildings correctly. For the jungle, we use the color orange to identify the jungle minions. A threshold was again used for this range of the hue histogram to obtain a detection criterion. A specific exception is two jungle monsters in the main diagonal, which were detected by using the same technique that was used for character identification. This is explained in detail in the "Identifying characters" section below. The histogram vectors of jungle monsters and ally buildings are shown in Figure 17 and Figure 18 respectively.

Figure 17. Jungle monster histogram

Figure 18. Ally building histogram

Hough Transform for minion disks

We apply the idea of a Hough transform for disks (see Figure 19). The accumulator space for the transformation is for disks of specific radius but of unknown location. This means that we are looking for (x,y) pairs that signify the center of our disk. For every point that is potentially a center of a disk that can produce the pixel we acquired from the minion image, we add a unit to our accumulator space on the disk's central location.

Since the centers of all disks that can produce our point form a disk, we simply add a unit disk centered on our current point to the accumulator. In the end, we are left with points of high value that represent disks and background noise is filtered out. As we can see the filtered image contains pixels with the correct hsv value. This value varies depending on whether the minions are enemies or allies. In this case we detected all the pixels with a certain saturation and value that were cyan (see Figure 19, left). After we acquired this filtered image we performed the disk Hough transform and produced the resulting Hough transform space (see Figure 19, right). We can see that the character's circle is visible but doesn't have a high enough value. We can clearly see that the minion gatherings have high values and can be

detected easily. Since we know the radius of the disk, after a valid detection we remove the peaks that occur inside the area of the detected disk. This means that inside every minion wave we find around 4-5 disk centers and can update our map state accordingly.

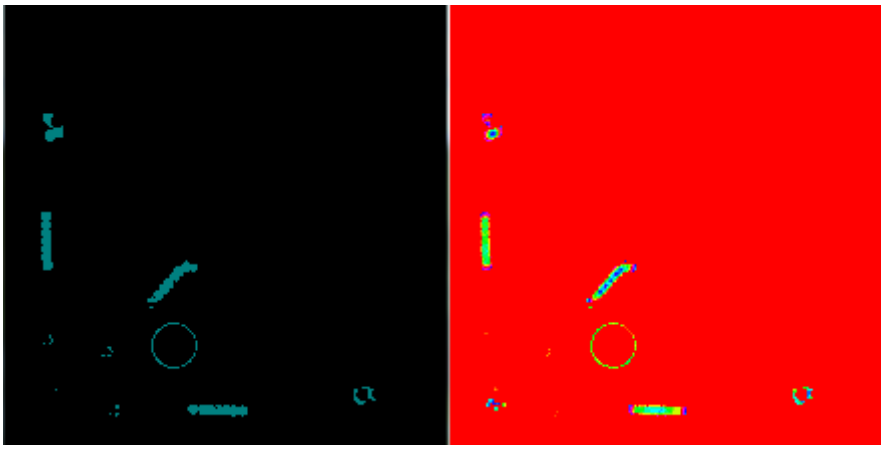

Figure 19. Filtered Image (left) and Hough transform for minions (right)

Hough Transform for character circles

To perform the Hough transform for character circles, we need to process the character/minion filtered image. We use a negative edge mask to detect where the image pixels go from cyan values to other values. After we apply the mask we get Figure 20, on the left, and we extract only the pixels in red color. Now we can use these pixels to apply the Hough transform method for circle detection. We simply have to calculate, for each pixel, the circles that are able to produce it. The Hough transform again has two parameters, (x,y), since the location is unknown but the radius is known.

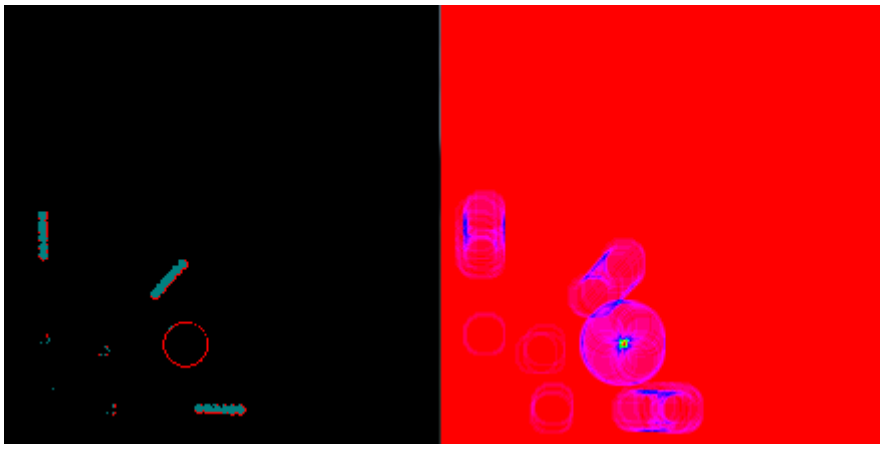

Figure 20. Processed Image (left) and Hough transform for characters (right)

Since the center of the circle produces all the points, we get the correct location of the circle in the accumulation plane (see Figure 20, right part). All the false peaks are created by circle parts formed by minions that simply do not add up. After we detect the circle we remove the center so that we do not detect the same circle twice.

Identifying characters

Figure 21. Character hsv vector

Identifying the characters is done through a simple hsv histogram (see Figure 21). We compare the Manhattan distance in the vector space between the currently perceived histogram in every component with the prototype. This vector was taken from a disk-shaped scan box which was applied on the resulting circle centers of the Hough Transform method.

Another way to detect the characters inside the map is using correlation hue vectors (see Figure 22). Every point in the image below exists in a two dimensional plane where every dimension is a hue vector. When the character's pixels jump from a hue value to another, we represent it by a point inside this plane. The horizontal axis refers to the hue of the original pixel while the vertical axis refers to the hue of the destination pixel.

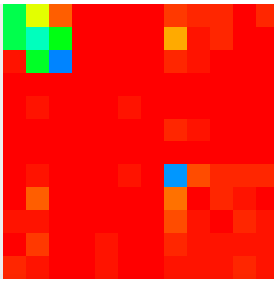

Figure 22. Accumulating correlation matrix

This way we can differentiate surfaces of many consistent fillings from surfaces with alternating regions of colors, providing means for more accurate character identification. These methods are used in

coordination to make sure our character identification is valid. Since they are not completely interdependent, their combination gives greater detection accuracy. In this case, the whole is greater than the sum of its parts.

Figure 23. Complete detection of map features

Detection of non-colliding features is 100% accurate (see Figure 23).

4.2 Visual Cues, the View System

Figure 24. Main game view with entities visible

The actual in game view provides enough information about the environment through a user interface filled with health bars and other statistics (see Figure 24).

After we focus our efforts on detecting lines consisting of black pixels, we acquire the location of important features. We need to preprocess the main view image to infer the locations of entities on the map. These include the character health bar and building or minion/jungle health bars, and unfortunately background noise as well (see Figures 25, 26 and 27). We process the main view and keep only the black

pixels. For the character health bar, which doesn't contain any black pixels, we filter out all but the grey pixels of certain hsv values. What we get from this filtering of the main view image is enough to locate all the necessary entities for robust recognition from the view.

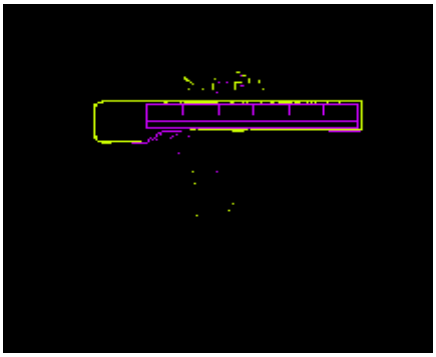

Figure 25. Character health bar

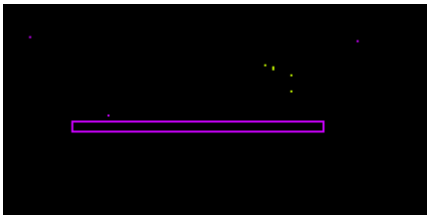

Figure 26. Building health bar

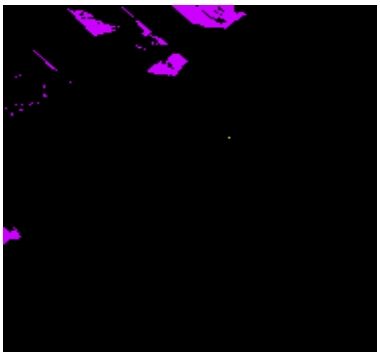

Figure 27. Background noise example

It is clear that there is a lot of noise surrounding the locations of important features and we have to remove it. We detect character health bars (see Figure 25) and building health bars (see Figure 26). Background noise is also detected but ignored (see Figure 27).

To actually detect the lines, we need to make a mask and a threshold that will filter out background details. First we obtain an image where every pixel is 1 if it is black and 0 otherwise.

We use the mask "D" to form an image that is thresholded for values larger than zero to produce the final image. Then we will be able to obtain line segments that we can classify as entity health bars.

Figure 28. Mask for line detection

If we apply the mask (see Figure 28), which is essentially an edge detection mask, and keep all the values larger than zero, we obtain the image which contains line segments from health bars only. Even the black filling inside a depleted health bar will not show up, which is precisely what we wanted. Afterwards, we detect lines of a certain length that are continuous.

For minion health bars, we remove duplicate lines that are spaced within a certain distance of each other and add to the collection of double lines that make up minion entities.

For characters, we first detect the health bar, and count the discontinuations that reflect a one hundred value increment on current health. We also assume the maximum amount of mana, which enables actions, based on character level and get a percentage of the blue line length of the mana bar, enabling us to estimate current mana.

The character below a health bar is detected based on their hsv histograms. We combine the hue, saturation and value vectors into one big description vector and compare with our character's prototype vector (see Figure 29). The scan box used to form the histogram vector is of a square shape, so that the character can fit most of his pixels inside it. Round scan boxes would take parts from the health bar and evaluate them as if they were character pixels so we don't use them. The vectors are normalized based on the number of pixels we detected. This is necessary because while in our map the amount of pixels visible is always the same, in the view, by removing excess pixels from the background, and based on character rotation, we obtain a variable number of pixels from the character's image (see Figure 30).

To actually compare the hsv vector to our prototype vector, we weight the components, so that differences between the different components won't affect the "distance" between the sample image and the prototype in the same way. Value was the most important component, so it was weighted by a factor of 3, saturation was weighted by a factor of 2 and hue was weighted by a factor of 1. Since this weighting was implemented before creating the prototypes, we can be sure that the same weights apply to the original prototype vectors.

Figure 29. Character hsv histogram vector

Figure 30. Image for hsv vector extraction

What we do to get a good ratio for character to background pixels is removal of the background that is low in saturation and brightness so we can get the true colors as shown in Figure 30.

We now turn our attention to the problem of finding where walls are. Figure 31 is a mosaic representation of map and view features. The walls are mapped from a two dimensional map data image to our 3d view. Features detected inside the view are placed without relocating them to a new place, since their original places are correct.

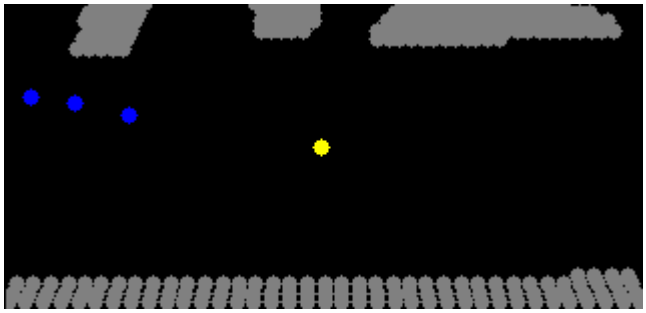

Figure 31. Wall transformation from map to view

Figure 32. View of the walls

Notice that the walls in Figure 31 (the grey dots) correspond to the actual walls in Figure 32, so that now our character may use manual pathfinding.

2d transformation

A transformation from a 2d image to a 3d perspective view is required. For this transformation, we first observe that the location horizontally affects how much stretching and shearing occurs while moving vertically. At the center there is no horizontal transformation, but as we move towards the sides, and closing in on the top of the screen, our location gets dragged towards the center.

By using observed and expected values for certain fixed points we determine the values of constants necessary for the transformation to be a good approximation. To do this we use points that have the same value in one variable, but a different value in another (for example, same x but different y). The variables do not interact with each other since we have a sum of products of at most one unknown variable each. Thus we obtain the correct values for the constants.

The walls were already in the minimap but not clearly defined. We used black pixels to represent walls from an image of the map, and white pixels to erase black pixels that were not part of walls (see Figure 33). Our software detects where the black pixels lie and places walls in those locations.

Figure 33. Wall data in map image

Finally we update the view state to reflect the walls inside our 3d view from the 2d to 3d mapping of the image's wall data.

Money OCR recognition

In order to observe the amount of money in the possession of our agent, we have to use optical character recognition on the image that represents the amount of gold we have.

An example of such an image is shown in Figure 34.

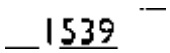

Figure 34. Sample money image

In order to extract the characters, we use a library that separates characters efficiently. Unfortunately, built in functions to actually identify the characters were available but suffering from accuracy issues. Thus, we trained a neural network that would identify singular digits from their images with a high degree of accuracy. The digits themselves show enough variation to confuse naïve methods for character extraction (see Figure 35).

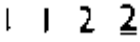

Figure 35. Separate digit images and variations

We train a neural network based on the Encog java library to make the detection of digits accurate. Instead of using pixel values inside the

images (which would require 400 input neurons for a 20x20 grid) we use a line-intersection method [7].

Initially, we produce a lot of lines on the images randomly. Then we evaluate how many of those lines were important based on a measure of entropy. If a line is touching a digit and is crossing around half of the digits, its entropy will be at approximately its maximum value and the line can be used for digit detection.

If a line is crossing most digits or none, it has low entropy and we cannot use its value to differentiate between the digits. Keeping a small number of lines was then used to train an artificial neural network on the samples of digits (see Figure 36, unique color for every line).

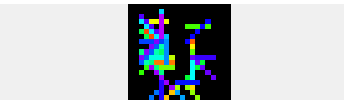

Figure 36. Example of scanlines used to identify the digits.

After enough variation samples were accounted for, the final classification was reliable and accurate. The neural network used three layers, one for input, one for output and a hidden middle layer using double the number of neurons of the input layer. After training the network, we used it on the cropped digit images to properly classify the digits. We obtained the final amount of gold by multiplying each digit with a proper power of ten. Since sometimes the OCR library omits detection of certain digits, we do not have a completely accurate detector, however this event is rare enough that every time our agent is trying to calculate his sum (around 5-6 times in every game) he will be able to calculate the amount of money reliably.

Skill Upgrade

We use the same methods that we used to detect jungle and building entities to detect whether the skill upgrade button exists. These buttons have a fixed location and appear only when our character has to upgrade his skills. This way we can keep track of our character's current level, which affects a number of game statistics, thus enabling us to accurately estimate character information like maximum amount of health or mana.

4.3 Decision Making

Items

Item selection is linear based on the amount of money we have. Every time our agent visits the nexus, he can buy more items based on his current gold. He spends the maximum amount of gold he can to buy the items in the list. Every item bought is subtracted in gold value from our total amount of gold. We stop buying items when we reach a total of 6 core items and there's no more space for purchases. Sometimes the agent sells starting items (currently, without further modification, just the "Doran's ring" item) to make space for more important late-game items.

Cooldowns

We have a specific order for upgrading skills. After our game state is informing us that there's a skill upgrade to be made, we press the right combination of keys to upgrade our skill. If this was the first upgrade of the skill, it enables us to use it for the first time. Another system, the cooldown calculator, is able to tell us which skills are available due to being upgraded. Using those skills locks them for a certain duration. We use the system's clock to measure the exact time it takes for a cooldown to end. After that time has elapsed, the skills are available for use again. An understanding of our opportunities for aggression is based on the exact amount of skills that are free of cooldowns. This proves really important when we try to remove the enemy champion from the lane. Thus, cooldowns are a measure of aggression and used in the "aggressive" mood state.

Strategist

The strategist is responsible for the formation and evaluation of interesting points in the minimap. It is tasked with choices pertaining to strategy.

The game itself forces the champions to have a starting laning phase, where they stay between enemy and ally turrets and fight for gold. Our champion ignores other lanes when the game starts, and after the first lane turret has been destroyed, becomes free to roam around the map.

The creation of interest points depends on our observations of the map state. Collisions of minions, team fights, enemy characters, minion

waves pushing turrets and even chat commands allow for the development of points in a list.

For every such point, the distance from our character is deducted from the initial interest value (which differs based on the event), as is the distance from the center of operations (which is the lane to which we have been assigned to). After the laning phase is over, it is only the distance from our champion that is accounted for. This permits free late-game movement to interest points.

When inner turrets near the nexus are attacked they get huge priority bonuses. But the most common event that our character is motivated towards following is minion wave collisions, where ally and enemy minions are in close proximity.

After the list of interest points has been created, we evaluate the interest points to find the one that is the most important by comparing its priority value against the other ones. This is sent as a signal of where to be in the mood system.

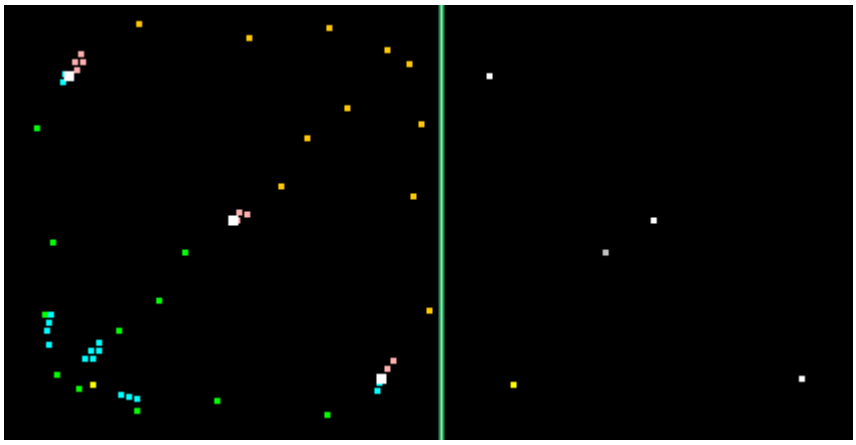

Figure 37. Strategic interest formation (left) and final interest point (right)

In Figure 37 we see the strategist in action. In the left part of the figure, we see the many auxiliary points that help with the formation of interest points. Green and orange dots are buildings while cyan and pink dots are minions. Depending on the kind of event that happens on a location we assign a priority to the interest points formed. The final points can be seen in the right part of the figure, with the yellow dot representing our character.

Mood

The mood system is a complicated system of behaviors centered on a theme. This will occur once for every perception-action cycle. This system decides to use only a small set of decision algorithms for every

kind of targeted mood. We get around three cycles every second meaning we couldn't manage a system that globally estimated every kind of decision simultaneously. Since we can make around three actions every second, we get ample of time to switch our behavior phase and behave differently. This proves effective and versatile in dealing with changing environmental conditions. The mood is developed based on our current estimates of the game state. Minions, turrets and enemy characters make our agent "Scared". If the view cannot detect our controlled champion, the mood becomes "Confused". If there is an enemy turret that we cannot really see but is close enough to our champion, the agent becomes "Curious". Enemy characters make the agent "Aggressive" while enemy minions make him "Greedy". If the agent has to go back he becomes "Homesick". Finally, if an interest point exists, the agent is "Interested". This system decides which kind of algorithms will run to determine our actions, based on the exact amounts of all these mood types that are competing for control.

After the mood has been developed, a comparison between different moods with different priorities ensures that the proper action type will be taken. However, even when we know the approximate action strategy the agent should be utilizing we still have to define a complex behavior based on the game state.

Mood state: Confused

The agent has use this action cycle to relocate himself. He centers the view on himself by using the spacebar key and continues to the next cycle. We don't want to detect anything else or take away precious processing time from our next cycle, so the visual perception process ends immediately.

Mood state: Curious

The agent is close to a turret that is not hidden by the fog of war (turrets outside ally minion/champion ranges are not available for detection from our view systems). We move the view towards the turret and change the state so the information is updated based on what we saw. This means that the game state will be able to affect what the visual phase of the next cycle will be able to perceive. This is not unlike eye movement where we notice something interesting in a place we do not currently see and move our view to perceive the point of interest.

Mood state: Homesick

The agent is low in health and has no other interest. He goes back to the nearest turret to recall back to the base. This is possible because we can estimate which allied turret is the closest to our agent and start recalling there. To accurately determine which places are deemed safe we use the map state.

Mood state: Interested

The agent has located a place of high interest. He wants to take an action to move to that place. The game already provides a robust algorithm for path formation from any point to any other. Our agent simply right clicks on the minimap to the place where he wants to be at. This is the mood of least priority and is only activated if other moods are inactive. To actually determine which place is of highest interest, we first form every kind of interest point on the minimap. These include but are not limited by minion collisions, character collisions, turrets being attacked by minion waves, turrets being attacked by characters, interest points created by chat commands and lane points (which are created after we choose the lane we want to play at).

Mood state: Greedy

When there are enemy minions on the map, the agent becomes "greedy". Since minions are the most important way of acquiring gold our agent is focused on last hitting as many as possible. Enemy minions have to die by our agent and not some other cause, to give their gold to him.

Our agent performs corrections on his position based on his ally minion wave (since being outside it means he can be attacked by enemy minions). If a low health target is available, the agent moves towards it. When an enemy minion is about to die, our agent attacks it to get its gold.

Sometimes the agent has to use skills (rarely because the skills require an expendable resource, mana, and then have a cooldown period where they cannot be used again) to destroy the enemy minion wave before it reaches our turret. Minion waves that reach a turret are quickly eradicated by powerful turret attacks, denying our agent gold.

Our agent's most promising way of dealing with an enemy minion wave is using a skillshot. A skill is called a skillshot if it is a projectile shot that takes skill to aim properly. When there is great variance in enemy minion location, determining the best line for a shot can be

difficult. Also, our champion has to target a point, so to make sure the skill follows a specific line the starting point has to be on the line too.

Our agent calculates the best line that fits the enemy minions' locations and then follows the shortest path to it. After he arrives at his starting point, he uses his skillshot towards a point on the line that is further away, towards the minion wave. To do this, we perform a regression fit of a straight line to the set of coordinates of the minions by using the Deming regression model. If enough minions are visible inside the view, and we have enough mana, we decide to use the orb attack to kill or lower the health of as many minions as possible.

Mood state: Scared

The agent is trying to find escape routes from sources of danger. We have used a mapping from two-dimensional wall data to three-dimensional obstacles. Thus, we have access to obstacle information as well as the sources of danger.

We use the A* pathfinding algorithm to follow the path of least danger. We make a grid of nodes where every node that is a viable path is connected. The weight of the path is changed to reflect the fact that moving close to danger sources is harder than moving away from them. Thus danger sources like turrets radiate their weights first based on distance, and when the A* algorithm creates the starting nodes it applies the "danger proximity" value to the distance between nodes. We find the fastest way to move from our current location to a safe side of the screen which lies near an ally turret by using the results of the algorithm. Doing so produces safe paths that do not follow the normal pathfinding provided by the game (which only takes distance in consideration) but also a utility cost modification that weights the danger of a path against the shortness of its length.

After we find the proper path, we follow part of it towards the edge of our view. We use the Manhattan distance to calculate the admissible heuristic of the A* algorithm, since we only make horizontal and vertical movements. This means that our heuristic will never overestimate the distance, since it represents the minimum distance possible.

Mood state: Aggressive

The agent has calculated whether he is in a winning or losing situation against a close opponent. He has decided that he wants to engage the opponent. Either he is a little aggressive and just wants to "poke" the enemy or he wants to go for a "kill". Poking uses a minimal set of

actions that do not expend a lot of the agent's resources. Trying to kill the enemy player results in loss of resources and is to be made after we can predict that there is a great chance it will lead to a kill.

The cooldown calculator gives a rough estimate of what our agent can do, and by comparing health and cooldowns we can approximate our chances for winning in a one vs one situation. A team fighting potential module also calculates whether we should engage in a larger fight that includes more than two champions fighting.

After all these imply the opportunity for attack, our agent employs a coordinated attack against the enemy. This attack puts certain skills on cooldowns by using them to attack an enemy character. After the coordinated attack is over our agent goes back to his ally minion wave because of the lack of cooldowns to use in attacks.

Chat Commands

We use a simple bag-of-words model to classify sentences based on their meaning [12]. Most of the work focuses on how to be able to respond in game, although there is no functionality to support transforming our answer strings into a series of keystrokes. However, certain mentions of objectives do interact with the Strategist to create new interest points for our agent, like the dragon, the blue buff or the baron jungle monsters.

For the model itself, we create mappings between certain words and a standard list of words. This standard list also creates mappings between the few words in the dictionary with concepts. Then we get an activation system that determines the way concepts interact with each other, activating more if they are related positively and less if they are related negatively. Finally, we check the activation status of our output concepts which determine what behavior our agents should have.

In case there is a chat command that says we should go to an objective, our agent is able to form an interest point in the minimap that will guide him there strategically. If he has no interest in pursuing something inside the view, he is inclined to follow the objective as commanded from an ally through chat.

Only one response is an actual interaction between the agent's actions and the chat, while others are simply ways for the agent to communicate in future implementations. The agent understands and forms a response without actually typing it, since we do not want to spend cycles typing instead of interacting with our game environment.

4.4 Action Mappings

To output actions to the game, our agent has to make simple output actions that the game understands like moving the mouse or pressing keys. To do this, we rely on an abstraction of the output system, which assigns general commands, like moving the view or clicking on a point of the map, to actual movements and clicks on the lowest level of abstraction.

To create input our agent maintains a queue of commands that are to be executed in priority. After we determine the abstract action we want to perform, like an exploration movement to Point a, the system first determines what mouse and keyboard actions are necessary. After determining this middle level representation of keyboard and mouse inputs, we convert the input to the lowest level of input. For example the action of moving the mouse and then pressing "Q" becomes a mouse movement to a specific point, a "Q" key press and a "Q" key release for a certain duration. After this is done, we consider the action completed and we can follow with another action.

4.5 Implementation

The following is a list of specific implementation details and imported libraries necessary for implementing the agent. The libraries are open source and free to use.

- The neural network we created used 50 input neurons and 100 hidden neurons to calculate the result in the 10 output neurons.

- We used the encog neural network library for creating and training the neural network for optical character recognition.

- We used the JavaOcr library for extracting characters and separating them in different image files.

- We used the Tesseract java library for determining the content of chat images, where players communicate with each other inside the game.

Chapter 5. Results

Firstly, we want to determine the accuracy of our visual systems. They prove robust enough to be used in isolation as an auxiliary api that the agent can use to acquire utility from the environment.

Secondly, we want to see if the agent can survive in the environment and have a decent performance. Verifying the exact effectiveness of our whole implementation will depend on comparisons to new human players, beginner bot agents as well as dummy "random" agents.

If we explore the idea of a random agent inside the game we soon come to find that it is going against our intuition of what we should compare against. If we take chess as an example, a random agent would produce meaningful choices every once in a while, making seemingly ignorant moves when something is obvious to a human player but completely ignored by the agent. This is not the case with League of Legends. Coordinating even a simple action proves much too difficult for a random agent. The degree of freedom is so vast that any kind of randomness in our agent proves detrimental to his success in the game.

5.1 Screen Capture

It was essential for the agent to be able to capture the screen's elements with great accuracy. The second most important source for information is the mini-map. When the different elements of the minimap do not collide, we get 100% accuracy in detecting them. This holds true even when the box surrounding the view (which is a white parallelogram) alters the values of the elements enough to distort their histograms. Consequently, wherever we look non-colliding map elements will be detected with complete accuracy.

Since features can possibly collide, we made sure to use techniques that are not susceptible to variation and distortion of the original features. When minions pass through the terrain and meet turrets, the turrets are for the most part recognized without problems. However,

when the character almost completely obscures the turret, it is hard to recognize it and it becomes deactivated. This is important because it shows that we do not get false positives if the element significantly changes.

Figure 38. Scanbox feature detection in the minimap

As we see in Figure 38, we get black and white boxes for turrets, which have all been recognized except the one that collides with our character, grey boxes for inhibitor buildings, orange ones for the jungle (as well as green for the dragon and baron jungle monsters), and finally cyan and pink for the minion waves. The minion waves are composed of many little disks that represent minions. Our view is shown in yellow, where the left, middle and right part have been recognized completely. The character is shown in blue, and enemy characters, if any, would be shown in red. To see how we handle false positives, we take another screenshot when some features are missing.

Figure 39. Scanbox feature detection in the minimap(cont.)

As we can see (Figure 39), false detection is not a problem. Every element that is missing (notably, most turrets in the main diagonal that runs across the river) is not detected by the screen capture system. Notice how even when the characters at the bottom of the image are really close, they are still recognized as separate characters that have been identified correctly.

In conclusion, false detection is not an issue. Even colliding elements of the image have a high chance of being detected properly. This is important because most of the time some kind of overlap will happen and no matter how good we can detect elements in isolation, not being able to detect them when they are close to each other would be a big drawback. Fortunately, the robust algorithms employed make sure to detect the elements when they are there and detect their absence when they are not.

The Hough transform idea on two parameters can also be shown to be accurate for characters (see Figure 40).

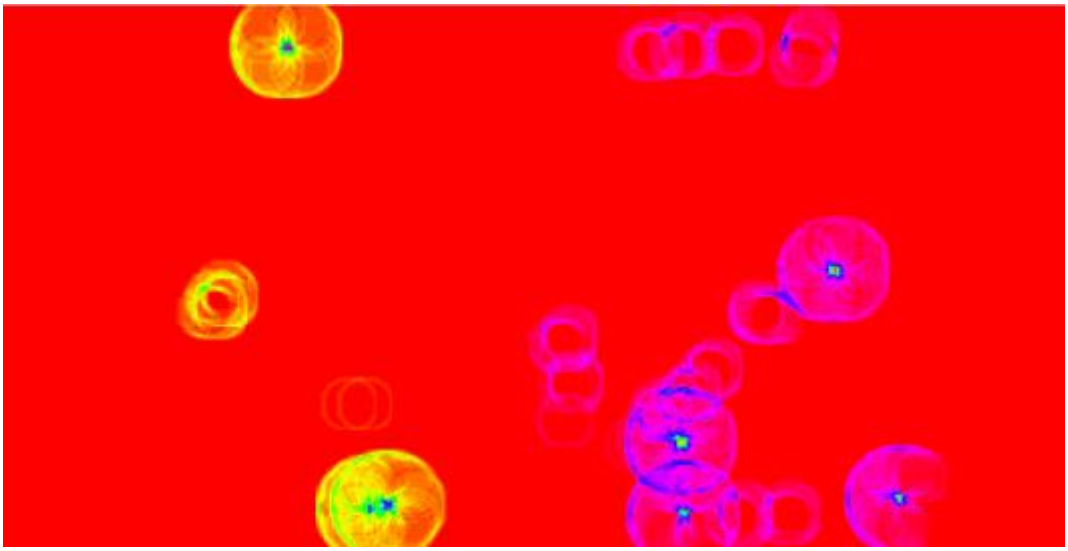

Figure 40. Hough Transform (for both teams)

The detection of characters is easy, because the peaks of the circles are way above the threshold of character detection, and minion circles, clearly visible in the center of the left part and the top of the right part of the image, cannot pass the threshold and prove they are characters. Before actually using the Hough transform, we had to make sure to remove the filling of the minion disks so they would not create a lot of possible circles and make detection harder (see Figure 41).

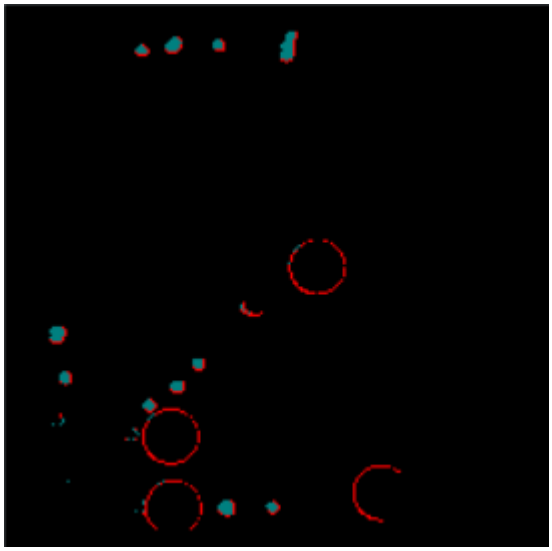

Figure 41. Edge detection on minions and characters

We are thus able to detect the circles by using the Hough transform for circles in this preprocessed image (Figure 41).

After all the map elements have been accounted for, it is time to move on to the view elements. This is the most integral part of the process. We need to be able to detect characters, minions, buildings and walls. The health bars must provide a means to get the current health of every such element, and provide clues as to what we can do to improve our utility income.

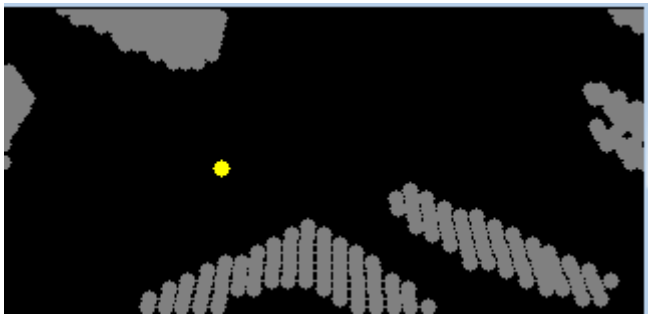

Figure 42. Walls (in grey) mapped from 2d data to 3d perspective

First we must detect where the walls lie in our view (see Figure 42). This figure shows a pretty accurate description of walls that is needed when we want to employ manual pathfinding. These have been mapped from the 2d map image to the 3d view plane. The yellow dot is our character, based on his position inside the main view, and the red/blue dots are the minions (see Figure 43).

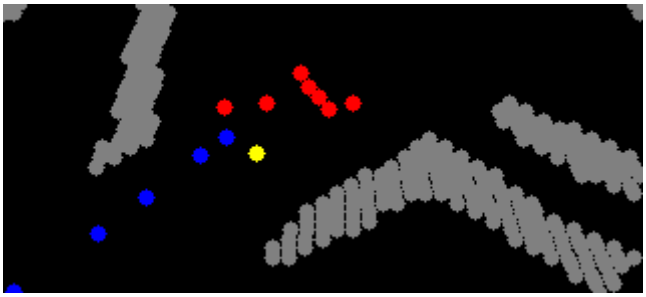

Figure 43. Minions (blue and red) , characters (yellow) and wall mappings (grey)

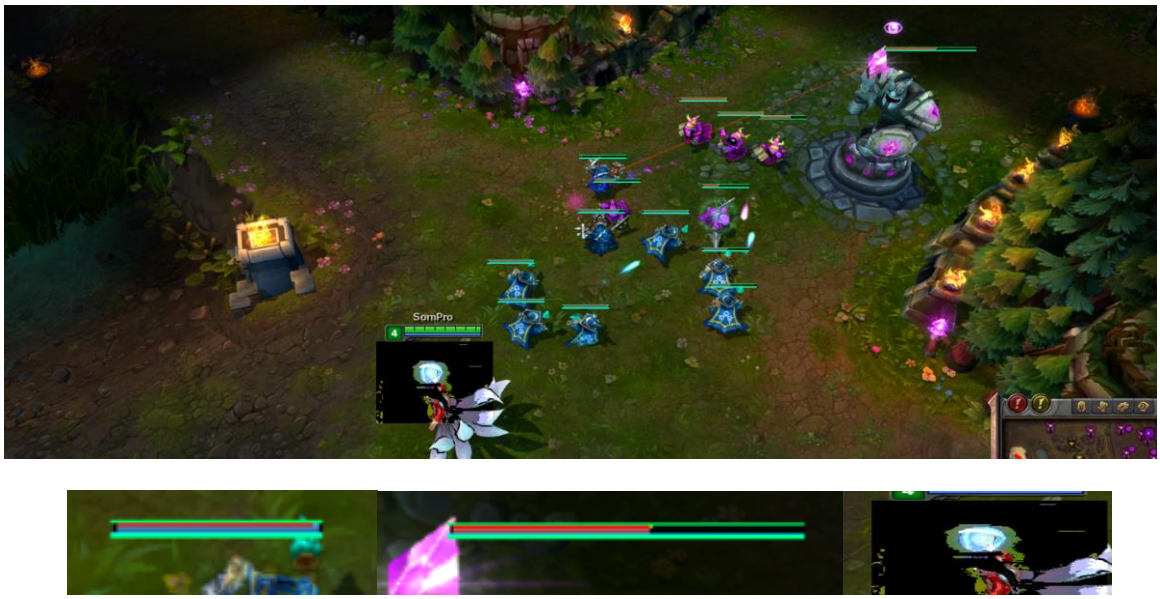

Figure 44. View element detection features

As we can see in Figure 44, we accurately detect the health bars of every element on the screen. This not only applies to the location of the entities but also on the amount of health they have. The red line that runs on the health bars is part of the algorithm and shows what percentage of the bar is filled. Unfortunately, the exact amount of minion health is not known because the bar represents a percentage. For the character, this is not a problem since the health bar is divided in 100-value increments, so we can get the precise amount of health. The character's maximum amount of mana (the champion's blue part of the health bar) is based on what his character level is, and given this maximum amount of mana, we get the approximate amount of current mana that can fuel skill actions.

Minion health cannot be estimated as an absolute quantity. Fortunately, since their health scales with our character's attack damage, we can safely assume that the percentage of health is as important as acquiring the absolute health of the minion.

This means that the percentage of health remaining qualifies as the only important parameter in estimating how many hits a minion should take before dying.

The bottom-right image in Figure 44 shows the character after all the background pixels have been edited out by our algorithms. This leaves the proper pixels that belong to the champion to be recognized. Character recognition in the view is not as accurate as it could be. This does not prove to be a problem because insofar no such information is used by the agent in decision making. We do not want to know which character the enemy is yet, but if we did want to know, since the accuracy is variable with time, we would want to have a system that would make reliable measurements by remembering which character was nearby from previous game states and correlate this information. The same issue does not apply to the map view because characters are not 3d models that can be rotated or obscured by background. In essence, the map shows a clear view of the champions involved and we also get the accuracy rates described in the map detection accuracy above (close to 100% accuracy).

Finally we make a statement about the accuracy of the money measuring system. The system has an accuracy rate of around 98% for detecting all the characters in the image. This is in part because the library we used does not always detect all the digits in the number and sometimes completely ignores some. This is rare, but it means our OCR will not be able to measure the number since every digit has to be multiplied by the correct power of ten based on position. Missing a digit will ruin this whole positioning scheme and change the whole number. Since we only perceive the amount of money we have on rare occasions (when we are at the nexus, around 4-5 times per game of 30 minutes), this does not prove to be a big problem.

5.2 Action Reels

To showcase our agent's behavior, we show some of the action types he performs when under certain perceptual schemes. For every type of behavior we analyze the cause for action (which is the whole behavior vector) and the result (which is a specific type of behavioral patterns directed towards a goal). When necessary we provide visual information that shows the agent in action.

First we have the action upgrades. If the agent notices that the buttons for upgrade are visible, he skips the entire cycle and simply upgrades the correct skill.

We then proceed to item shopping. The agent notices the amount of gold he has, and if he is in the base he proceeds to buy the correct items from the in-game shop.

Figure 45. In-game shopping tab

Here in Figure 45 we see a clear screenshot of the in-game shop function. Our agent uses the standard page that is recommended for our champion. If the game changes this order, we would have to use custom item sets, a functionality provided from inside the game, to be able to buy items in the correct order. A complete overhaul of this system could make it type the name of the item. We assumed this process would be prone to errors, such as delays in the keypresses that occur under heavy network load, and decided against it. It would also be time consuming to type every name letter by letter and we instead opt for fast sequential right clicking of the items.

When the agent is in the "interested" mood, he clicks on the map to travel to a distant point on it (see Figure 46). He has already figured out where to be and uses the game's built-in pathfinding (that works on the minimap) to travel to a certain location.

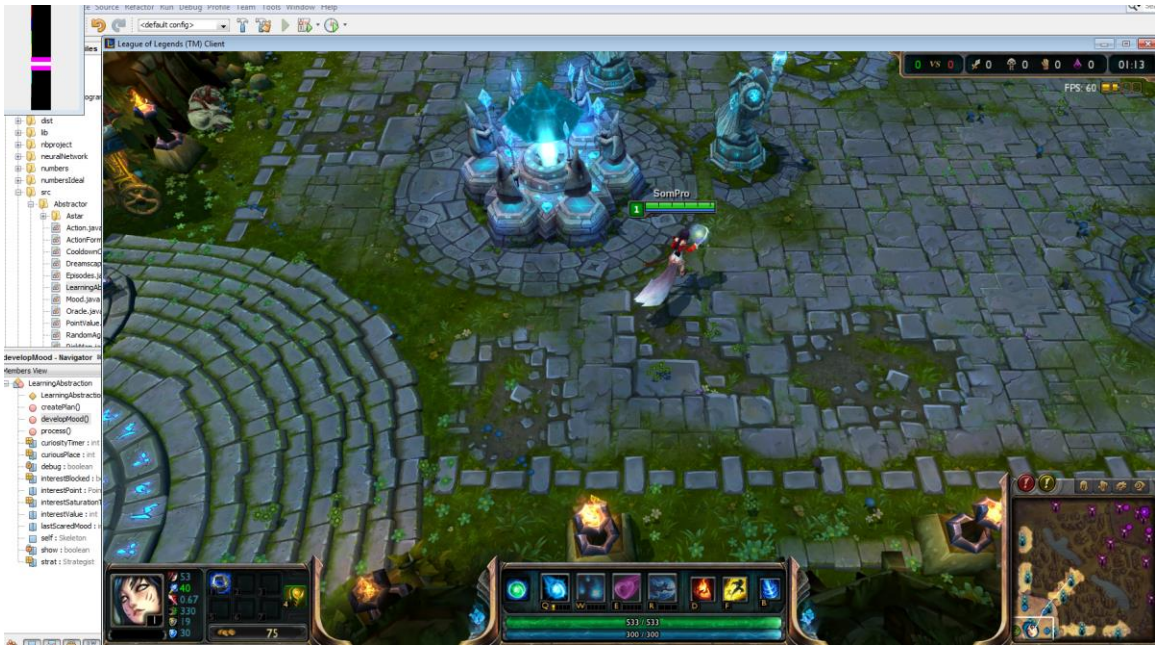

Figure 46. Mood "interesting"

Begin by noticing that the top-left part of the image shows the agent's moods that compete for actions. The current mood is "interested". The agent already clicked the point on the map that he wants to go to and travels towards it (bottom-right part of Figure 46 shows the line of traversal).

When the agent is in the "greedy" mood, he tries to kill minions by hitting them at the last moment to get their gold.

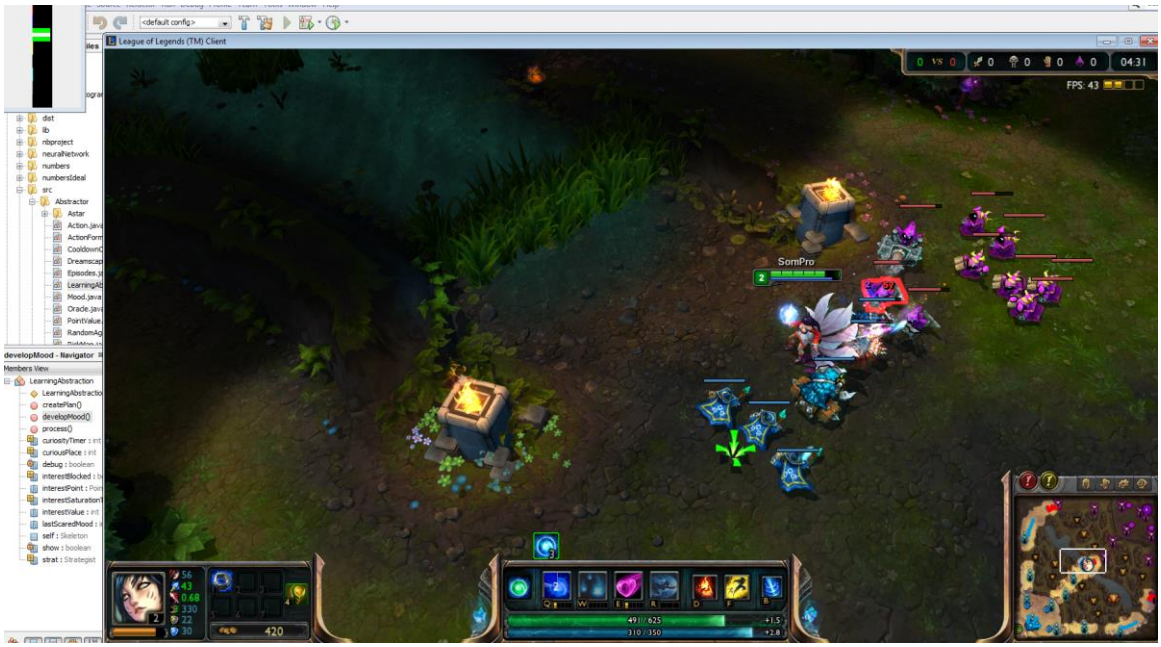

Figure 47. Mood "greedy"

This mood is represented by the color green. The agent either “orbs” (the main damaging action) the wave if big enough, or simply tries to hit the lowest health minions by moving close to them and performing the attack animation (Figure 47).

Figure 48. Agent using the "Orb of Deception" skill

Our agent is using his skill “Orb of Deception” to lower the health of enemy minions (see Figure 48).

Figure 49. Agent prepares to "autoattack" the enemy minion

Our agent is using his “autoattack” animation to last hit the enemy minion (see Figure 49).

When the agent is in the "confused" mood, he simply repositions the view on himself by using spacebar. This happens when the agent cannot see himself.

When the agent is in the "scared" mood, he moves away from sources of danger based on manual pathfinding using the A* algorithm. The mood color used is blue, as shown again on the top-left part of the image (see Figure 50).

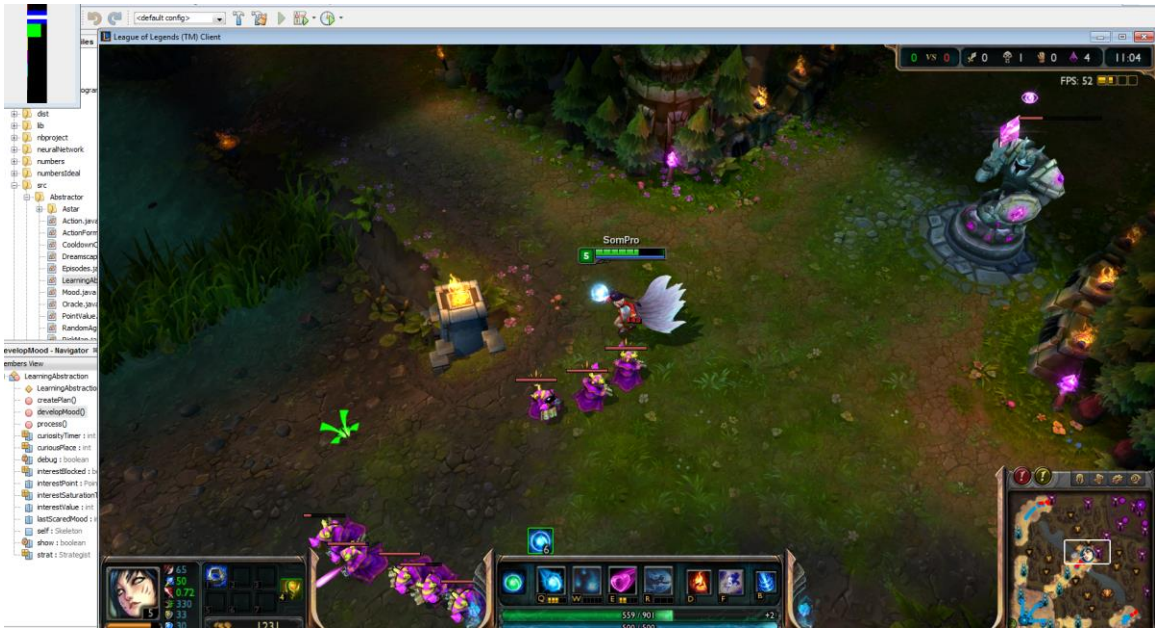

Figure 50. Mood "scared"

When the agent is low in health he simply prefers going back to the base to heal. The mood color is cyan (see Figure 51).

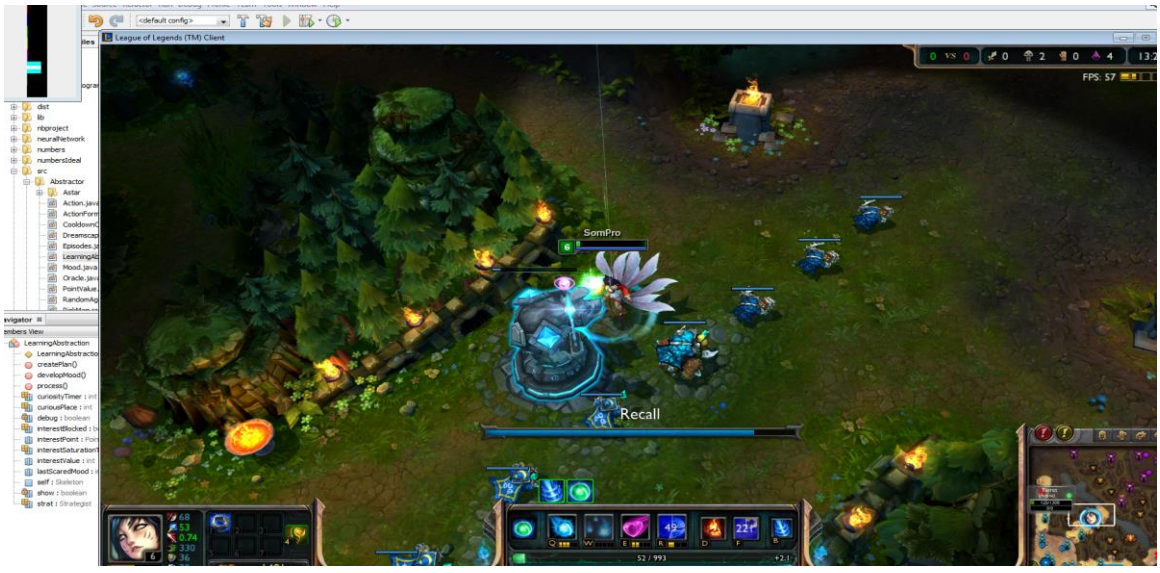

Figure 51. Mood "homesick"

The last mood is “curious” and it is used when we are near an enemy turret and want to know whether it is there or not. We move the view on the turret and try to detect it. The new information we acquire is updated in the game state and we continue with our next cycle (see Figure Set 52).

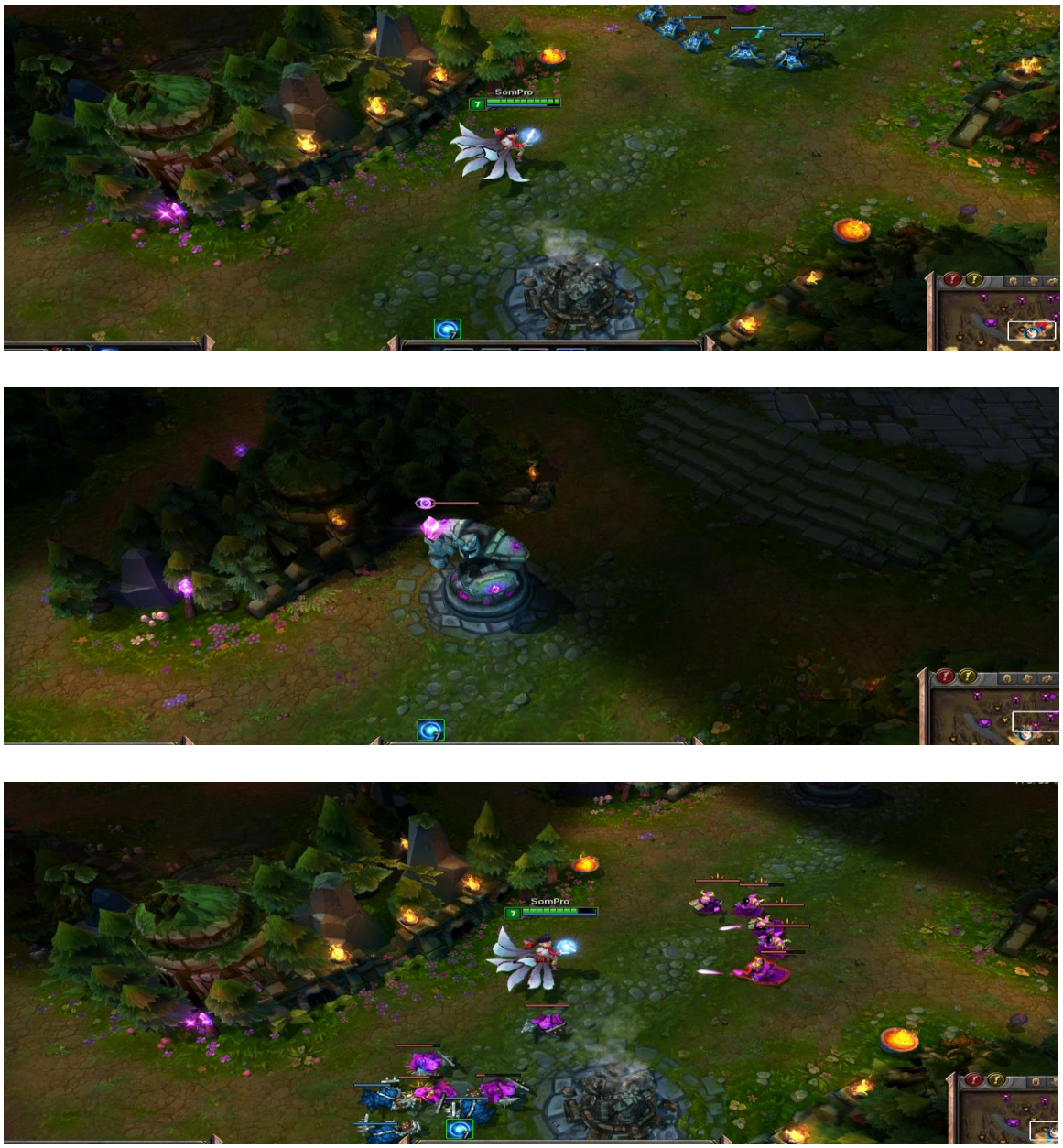

Figure 52. View relocation based on perceived turret threat

We can clearly see that the game state affects the next cycle by changing what the visual processing system wants to perceive. This also happens when the character is missing from the main view and we reposition it have him at the center. This cycle is very important in

versatile autonomous agents that can control their visual field's location or orientation (as in real world applications).

5.3 Random Agent

We start implementing a random agent to see how well he fares against the environment. Even after correcting the probabilities of the actions, we see the following results:

First, by using random move actions inside the minimap, we take advantage of the built-in pathfinding algorithms of the game, but since movement is completely random, with all locations being of equal probability, we finally reach coordinates (110,110), which means that the expected position of our agent as time approaches infinity is the center of the map (of size 220x220). After the agent reaches that place, and minion waves start to spawn, the agent is attacked and dies, unless already protected by his own minion wave. Randomly using his actions inside the game also proves troublesome since his main resource for using actions, his mana, is depleted quickly, preventing further action from being taken. Randomly clicking on the view of the map does not work when the agent is blind as to where the enemies lie. After running the software for 5 minutes, we see the agent approaching the middle of the map, hitting only one minion by accident and then dying. Before he reaches the middle of the map again, the time is up.

We conclude that the random agent is completely unable to perform any kind of meaningful action, so hard coding certain behaviors was correctly identified as the best way to implement our agent's core behavior patterns.

5.4 Dummy Agent

We now focus our attention to playing without an opponent but utilizing both our visual fields and playing properly. Our agent is running at his full potential, however since no opponent is present, we have no use for aggression patterns based on the presence of enemies. The main problem here is that our minion waves start building up and the damage on enemy minions is too much for our agent to timely last hit the enemy minions. That lowers our highest possible creep score (the amount of minions our agent has last hit for the game's duration). However, without pressure from an opponent our agent is able to stay in the lane longer.

Our scores for this game mode are seen in Figure 53.

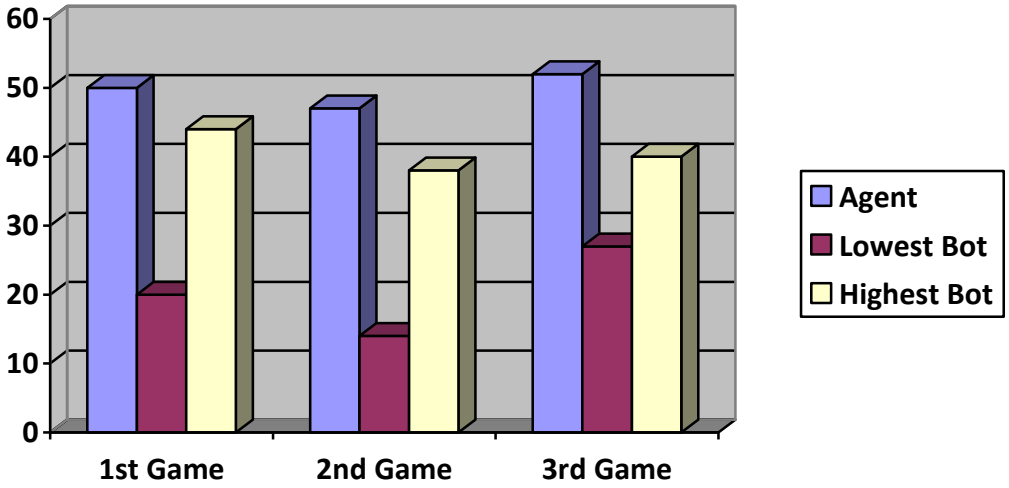

Figure 53. Agent comparisons for 1 vs. 0 situations

Our agent was able to beat the in-game beginner bots in 1 vs. 0 situations (where no opponent was present in lane). This means our perception-action cycle is fast enough to be able to compete with the scripted bots that use the API directly. For comparison, the ideal amount of minions a human player is expected to get before he becomes a game expert is around 70 without opponents. 70 minions is also the amount of minions expected of very competent players in a 1 vs. 1 situation. 50 minions per 10 minutes of playing the game is above expected for our agent and means the system handling our agent's last hitting is working efficiently when not disturbed by the presence of opponents.

5.5 Duelist Agent

Our efforts are now directed towards dueling with one enemy opponent. This means that the opportunities for strategic decisions are minimized and what we have to do is be tactically correct to be able to stay in our lane and minimize the time we need to spend away from the enemy minions. Doing this requires constant pressure on the enemy, with the ultimate goal of either defeating him or pushing him out of the lane.

When we duel with a lane opponent, we get less minions than when we were alone in the lane. However, we manage to remove the enemy lane champion from our lane and lower his ability to get minion remarkably. We remain ahead by around 10-20 minions from our opponent and we manage to obtain 1 enemy champion kill in the first ten minutes. Since the laning phase is usually over after ten minutes we stop our comparisons here. We manage to get around 40 minions every 10 minutes in the game.

Human players in the lowest league (meaning beginner level human players) score around 50 minions the first 10 minutes. However meaningful comparisons can be made only after realizing that most of the time, dueling with enemies in competitive play means a player will usually have the same amount of minion kills as his opponent. By managing to zone the enemy champion out of the lane, our agent creates a positive gold advantage by lowering the enemy champion's minion kill potential.

5.6 Complete Agent

Finally, we try an X vs. Y situation, where our agent is battling with allies against a multitude of opponents. This is the normal mode of the game, with a 5 vs. 5 being the common way of playing it. Our agent has to be better than his opponent to be able to have a chance at winning. After running certain games against the beginner ai agents of the game, we see the following:

First our laning phase worked in the same way as describe in the previous section. When the turrets started falling, our agent started visiting other allied champions and helping get other turrets. When team fights started happening, we had a higher chance of defeating enemy champions and proceeding with acquiring even more objectives like turrets. The gold advantage started piling up and we managed to win the game. This result isn't completely consistent because there are a lot of problems that can be magnified from the early phase of the game. These kinds of problems originate from lack of a completely accurate game state. Certain actions cannot be validated by perception and we have to assume they were performed without mistake. For example if our agent tries to upgrade a skill but a lag spike prevents the action from being realized in the game's world, we will have problems for the rest of the game. We discuss the implications of these problems in the future work section.

Chapter 6. Conclusions

6.1 Outcome

The first results we acquired from the perception module of the agent were relevantly inaccurate at best. Tinkering with specific algorithms and techniques to improve accuracy turned out to be harder than expected, highlighting the difficulty of integrating information about objects from many modes of perception. Implementing improved algorithms could prove fatal to the execution of the program under such limited spatial and temporal resources.

The agent does not learn new things for the time being, since without a good model of the game before learning, the results would not be sufficient.

Finally, the agent's software is comprised of more than ten thousand lines of code, but seems to be a really weak player compared to advanced human level play. This means that without a better machine, emulation of perception and action without learning would not be able to impress a panel of judges very easily. However it is important to note that the level of sophistication in agents that can beat human level opponents in most games is actually overshadowed by the brute force algorithms employed on a very abstract game state. This kind of agent, one that has to perceive a game from a snapshot, cannot enforce such methods since there is not an accurate and simplistic abstract representation of the game, with clear-cut rules and a small amount of moves available.

It is obvious that solving this problem efficiently would possibly prove to be an AI-complete problem that depends on cunning tactics, estimation of the opponent's mental state, learning new tricks by exploration and balancing utility vs costs in a very undefined and unstable game environment. With the advent of new computers maybe this will prove to be a much easier problem, one that will still require intelligent effort to solve efficiently.

6.2 Discussion

In this thesis, we provided a way of implementation that is different from the standard methods for dealing with game artificial intelligence. A more human-like approach was given that determined the amount of processing necessary to abstract an approximate game state, and program an agent that could play the game reasonably well. By using a standard home pc, we show that currently the processing power available to computers is enough to use a variety of algorithms to solve such a complex task.

It seems obvious that with more processing power we could employ state of the art algorithms for detection and reach a much better understanding of the game state. Currently, the perception of enemy actions, with the exception of positioning, is virtually non-existent. Without a good model for enemy actions, and without enough time to react to or predict those actions, our agent will not be able to compete against higher skilled players. With faster processing times we would be able to also implement machine learning algorithms that would be able to learn complex maneuvers that are currently impossible to perform.

Making an efficient agent for this game proves really difficult without accurate ad priori knowledge of the complete game state. If an application programming interface (API) is presented, this would make the creation of an agent much easier, but would defeat the purpose of making a human-like agent that uses techniques similar to ones in real-world applications.

6.3 Future Work

For the most part, the agent is not complete. The most basic component missing is learning based on exploration, and a second, also important part is learning how to estimate opponent level of play. If the agent is able to learn how to better utilize the environment without semi-scripted behavior, he will be able to really reach higher level of play without guidance. Estimating the opponent's level of play is also vital to being able to employ basic strategic patterns that will either defend against better opponents or be aggressive towards weaker ones.

If the agent's software is not able to execute on a faster machine, these would prove to be very unrealistic goals since any benefit the agent would gain from advancing his understanding of the game would be counterbalanced by the mere fact that the basic action cycle of roughly 300 milliseconds would be forbiddingly augmented.

Certain problems can cripple our agent's performance. Revalidating his actions through perception would be a necessary implementation for future development of the software. We would need to improve our skill upgrade and item purchase systems so that they work even under a heavy network load that causes some actions to not be registered in the game. Problems like these need to be eradicated in order to have a good foundation for a robust autonomous agent.

6.4 Lessons

The amount of work needed to complete this thesis made it possible to understand the kinds of problems involved in large project management. The coding project itself was more than ten thousand lines in length without counting revisions and corrections. To manage this kind of complexity we needed a plan, a good understanding of the object-oriented programming paradigm and a lot of patience. Error correction gave a deep understanding of debugging features and an overall insight into what kinds of problems can be caused by what types of coding errors. What is often said in programming is that there has to be a flow of ideas that can be translated easily into code. This demanding project helped train this exact kind of behavior where an understanding of the problem caused a flurry of ideas to come forth and be translated into code. The algorithms required in both visual perception and decision making made it possible for me to develop a strong knowledge foundation for future work. It also gave me an understanding of the complexity, spatial or temporal, of a variety of processes.

All in all, it was a character molding experience that helped me gain confidence in dealing with any kind of problem, whether complex or simple, large-scale or quick to solve. For the most part, it is sufficient to say that I learned what I was able to do with concentrated effort, making me trust myself and giving me a foothold for future reference.

Bibliography

[1] “League of Legends” game, http://gameinfo.eune.leagueoflegends.com/

[2] See Wikipedia, Multiplayer online battle arena,
http://en.wikipedia/wiki/Multiplayer_online_battle_arena
(describing the term Multiplayer online battle arena) (as of Sep. 5, 2014, 15:16 GMT).

[3] “Ms. Pac Man” Contest, http://cswww.essex.ac.uk/staff/sml/pacman/PacManContest.html

[4] W.E. Deming. Statistical adjustment of data. Wiley, NY Dover Publications edition, 1985

[5] P.E. Hart.; N.J. Nilsson; B. Raphael, B. "A Formal Basis for the Heuristic Determination of Minimum Cost Paths", IEEE Transactions on Systems Science and Cybernetics, 4 (2), pp. 100-107,1968

[6] N.J. Nilsson. Principles of Artificial Intelligence. Palo Alto, California: Tioga Publishing Company, 1980

[7] A. Kirillov , Optical Character Recognition,
http://www.codeproject.com/Articles/11285/Neural-Network-OCR, 2005

[8] H. E. Foundalis, “PHAEACO: A COGNITIVE ARCHITECTURE INSPIRED BY BONGARD’S PROBLEMS”, Indiana University, 2006

[9] R. C. Gonzalez, R. E. Woods Digital Image Processing, , Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA ,2001

[10] L. Shapiro and G. Stockman. "Computer Vision", Prentice-Hall, Inc. 2001

[11] D.H. Ballard, "Generalizing the Hough Transform to Detect Arbitrary Shapes", Pattern Recognition, 13 (2), pp.111-122, 1981

[12] H. Zellig . "Distributional Structure". Word 10 (2/3): 146–62, 1954

[13] M. Riedmiller , H. Braun: Rprop - A Fast Adaptive Learning Algorithm. Proceedings of the International Symposium on Computer and Information Sciences VII, 1992

[14] Rgb model, Mathworks Documentation, http://www.mathworks.com/help/images/reducing-the-number-of-colors-in-an-image.html

[15] M. Horvath, Hsv model, http://isometricland.net

[16] L. Stowasser, Neuron model, http://www.phpclasses.org/blog/post/119-Neural-Networks-in-PHP.html

[17] ISDA, University of Illinois, Urbana-Champaign, Automated Extraction of Areas of Interest from Scanned Documents, http://isda.ncsa.uiuc.edu/drupal/content/bicentennial-celebration-abraham-lincolns-birth

Appendix I. User Guide

This is a list of information necessary to be able to use the agent with the game.

- In the game options, we enable "smartcast" for all the actions our champion can take (called skills), and run the game in a windowed 1600x900 resolution

- We set the items page to "Recommended" (should be automatically there in case the account has not changed item set settings"

- We use the champion named "Ahri". We can extend the program so it can accommodate other characters, since this specific champion if one of the hardest to play with, but we are currently only using this champion, as he has a difficult skillset to master with "skillshots" and positional elements being the most important.

- We use the normal skins for all the champions and we get the "Ignite" and "Flash" summoner spells in the champion selection screen.

- The map and view detection of characters can be extended easily. Currently, the usual opponents are the champions named "Annie", "Lux", "Wukong", "Ryze" and "Renekton". The agent will not recognize enemies that are not part of the above selection. We can make the agent ignore enemy champion identification and just consider all opponents equally but that would rarely lead to misrepresentation of enemy minion waves as champions.

- We use the load from scratch option provided, and choose a lane before pressing start. It is advised to press the "Activate" button when the view behind our agent's software window is visible. If the agent does not capture the game view immediately, there might occur problems with the long lasting detection of map features.

- We use the button "N" to start or stop the agent from being active. Since the agent uses buttons without awareness of whether the game window is actually focused, we run into problems if we try to stop the program from the ide (integrated development environment we run the software from).