CS7032 AI Project: Developing a Vindinium AI Agent

Brian Maguire
10366921

Carl O Connor
10352941

January 26, 2015

Abstract

This report describes an approach taken to developing an artificial reactive agent to play the game Vindinium. The approach taken simplifies the game into a finite number of game states and actions each with their own associated reward value. A Markov Decision Process is used for decision-making based on expected rewards. Unsupervised reinforcement learning was used to update reward values. Over the course of a large number of games the bot showed significant improvements in performance. However we discuss some problems and shortcomings in the approach taken and suggestions for future work.

1 Introduction

”The development of full artificial intelligence could spell the end of the human race”- Stephen Hawking[1]

In recent years artificial intelligence has emerged as one of the major areas of research in computer science. From its beginnings in academia, it has quickly found a place in commercial applications such as AI game development. Today many classic games that were once played by humans, are now played by machines. However Vindinium is a game that was developed purely for AI to play.

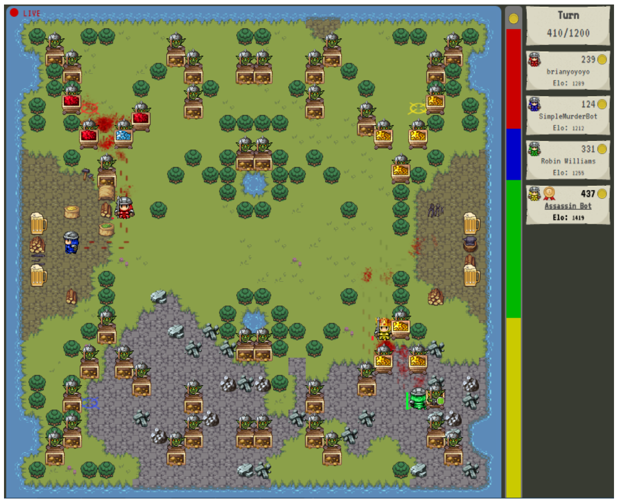

Figure 1: A Vindinium match [2]

2 The Problem Specification

2.1 The competition

For this project we were tasked with creating an artificial agent to play the game Vindinium [3]. Vindinium is an online AI programming challenge. Users register a username and can then create and develop their bot in any language of their choice. Bots can then be used to compete in online games; multiple games can be entered at the same time and multiple iterations of a bot may enter the same game. Game maps are generated randomly but are symmetrical so as not to give any bot a starting advantage. Games are continuously held and points are assigned for winning and losing games. A ranking table is maintained of the all time top performing bots.

There is also an offline training mode provided which does not affect rank and can be used for development and AI learning purposes.

2.2 Game rules

The main game rules for Vindinium are as follows:

- The aim of the game is to gather as much gold as possible (or more than any of the other 3 players).
- Gold is gathered by controlling mines.
- A player earns 1 gold per turn for each mine under his control that turn.
- Mines can be captured by attacking them or by killing other players. You take control of all of another players mines by defeating them in battle.
- When attacking a mine, a player must defeat that mines resident goblin, and loses 20 health points (HP) in doing so.
- Each game is broken down into 1200 turns.
- Each of the 4 players has 2 seconds to make a move, (Stay, North, South, East and West), each turn.
- Players spawn (and respawn after death) with 100 HP. This decreases by 1 per turn until the player has 1 HP remaining.
- Players can regain 50 HP by going to a tavern and spending 2 gold.

Figure 2: Tavern (left), Hero (middle), Mines (right)

2.3 Environment Properties

The Environment of Vinidinium can be described with the following properties.

- **Fully Observable:** The agent receives all information on the game after each turn.
- **Stochastic :** The next state of the game is determined in large part to what action the enemy agents choose to perform.
- **Sequential :** The game state, mines captured, hit points remaining, change over the coarse of a game, making the agents next move depend on previous moves.
- **Discrete :** The Game map is made up of a finite number of tiles, and values such as hit points and gold all have discrete values.
- **Multi-Agent :** The game includes 4 separate agents, it is possible for two agents to work together to win a game.

2.4 Agent Properties

The bots that play vindinium may be controlled in a variety of ways, but they all share some properties.

- **Performance Measure :** The agents may measure their performance based on gold collection rates or over a longer time frame, the leader board on Vinidinium.
- **Environment :** As stated above, the environment of Vindinium includes a randomly generated map and 3 other enemy bots.
- **Actuators :** The Bots may only interact with their environment by sending direction of travel.
- **Sensors :** The Agents receive all information through a JSON file.

3 Possible Approaches

3.1 Path Cost

The Vindinium map could be looked at as a far more elaborate version of the Grid World Game described in Lectures[2] and shown in figure 3. In this approach, mines, taverns and enemy agents are all seen as points of interest that are either favourable, or unfavourable.

The complex game is broken down into the simple choice of paths of different value. The values would be based on properties such as distance, hp for enemy agents or our own hp when looking at a tavern. These properties must be fed into a cost function, which outputs a number, either positive or negative. The bot then takes the direction which has the minimum cost associated.

This approach is similar to that of Potential Fields, used in real time strategy games[5]. Here objects emit a field across a map, getting stronger the closer one gets to it. This approach has the advantage of simplifying the next move to the road of least resistance. It is also limited in this, as the sometimes complex designs and set ups of a Vindinium game state must all be described by one cost function.

We briefly implemented a version of path costs but soon found our cost functions to be unsuitable in many situations.

3.2 Genetic Algorithms

Used in conjunction with the path Cost method, Genetic Algorithms can be used to tweak a system. To use it with Vindinuim, the cost functions used in Path Cost would have weights added to each component. These weights would stand in for chromosomes and could be mutated to produce an agent with different characteristics. The new agents can be evaluated based on their performance in the game. The changes that result in improved performance(more victories) can be crossed with each other to produce one agent with the best characteristic. The process can then be repeated [6]. We felt this approach would be too time consuming. Each iteration would need to play several games to determine weather the changes had been beneficial or not.

3.3 Reactive Agent

A simple solution to Vindinium is a reactive agent, which reacts with fixed rules, depending on different inputs. This method is requires a human to pinpoint the best strategies and to hard code the agents to follow them when they can. For example, "capture mines until health low, then go to tavern". We felt this approach would leave the bot susceptible to being taken advantage of, if it was in a substitution that was not coded for.

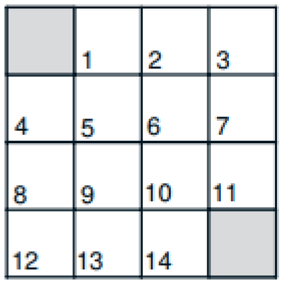

Figure 3: a simple Grid World [2]

4 Our Implementation

We choose to implement our bot using Ruby.

4.1 Decision Process

Our implementation involved using a Markov Decision Process (MDP) to assess the current game state and possible actions that could be taken and associated them with an expected reward.
A MDP is a discrete time control process where, at each time step, the process is in some defined state S, with possible state actions A, and corresponding action rewards R. The process enters the next time step by choosing an action A and moving into the next state S and returning the corresponding reward R. The system chooses the appropriate action, A, to take in each state based on a predefined decision policy. Decision policies can be optimized, to maximize reward returns, or stochastic, to encourage exploration of action rewards. This introduces the concept of exploration vs. exploitation in decision-making and machine learning.

4.2 States

Given the almost infinite number of states that could exist in this particular game we defined a range of states to simplify the decision process. The games current state was taken and functions were used to quantify this into a finite number of game states that we could process. Game state parameters that were of interest in defining the overall game state were those related to objects of interest. I.e. mines, taverns and enemies.

1. Our HP (dangerous, low, medium, high)
2. Our wealth (low, medium, high)
3. Nearest enemy HP (dangerous, low, medium, high)
4. Nearest enemy wealth (low, medium, high)
5. Nearest enemy distance (very close, close, medium, far)
6. Nearest mine distance (very close, close, medium, far)
7. Nearest tavern distance (very close, close, medium, far)

This resulted in 4*4*4*4*4*3*3 = 9216 possible game states overall.

Each action within a state was associated with an estimated reward. All the states, actions and associated rewards were stored in a hash table to allow for quick and easy access. The key to the hash table was generated as a string of concatenated characters, each describing one of the 7 game state elements. E.g. **'hlhmcmf'** would indicate:

1. Our HP = high
2. Our wealth = low,
3. Nearest enemy HP = high,
4. Nearest enemy wealth = medium,
5. Nearest enemy distance = close,
6. Nearest mine distance = close,
7. Nearest tavern distance = far.

4.3 Actions

For each of these game states there were 3 possible high-level actions that could be carried out:

1. Go to nearest mine
2. Go to nearest tavern
3. Go to nearest enemy

For this approach only the nearest mine, tavern and enemy were considered so as to simplify the decision process. Similarly it was deemed that it would be a waste of valuable turns going after objects of interest that were further away from the player.

These high-level actions had to be developed from the 5 basic commands that could be given to the bot (Stay, North, South, East, and West).

4.4 Episodes

To allow for episodic learning, the continuous game of Vindinium was broken down into episodes by describing termination points. These points were :

- The capture of a mine
- The death of the hero
- The killing of an enemy
- Visit to a Tavern

If any of these points were reached the current episode was over and a new one was begun.

4.5 Path Finding

Implementing a path finding algorithm was an important part in translating the basic agent actions into high-level actions we could use in the decision making process. There are many algorithms commonly used for path finding in gaming but one of the most common is the A* path finding algorithm.

An A* path finding algorithm is implemented as follows:

1. A list of nodes that the agent can move into from the current node is generated.
2. These nodes are added to a list of open nodes to be considered and linked to their parent node.
3. The parent node is then added to a list of closed nodes.
4. The algorithm then moves the current node to the node in the open list that is closest to the destination.
5. Steps 1 4 are repeated until we encounter the destination node and the path is traced back through all the parent nodes.

Our bot carries out this path finding algorithm each turn on each object of interest and considers only the mine, tavern and enemy with the shortest path.

4.6 Decision Policy

Our bots decision policy was based on the results of the N-armed bandit example in [4]. This example tackled the issue of exploration vs. exploitation in deciding between 1 of 10 actions to take in a simulation with the aim of maximizing returns from the system. It was found that an exploration factor of 0.1 (or 10%) provided the system with a good balance between exploring actions and exploiting rewards. Thus our decision policy was based around exploiting the optimal action reward 90% of the time and choosing one at random the remaining 10%. Modifying the greed parameter in the code could change this value.

4.7 Learning

Our bot carried out an unsupervised reinforcement-learning algorithm and used Monte Carlo policy evaluation to estimate expected rewards for value action pairs. These action value pairs were initialized to arbitrary values and updated based on episode outcomes. The updated values were saved on a JSON file after each game.

4.8 Reward Propagation

At the end of each episode the state actions pairs which made up that particular episode were rewarded based on the outcome of the episode. The rewards were as follows:

- **Mine Captured** $= 5 - \#ofsteps * 0.1$
- **Enemy Killed** $= 10 + \#ofminescaptured$
- **Restored Health** $= 5 + \#ofmines$
- **Died** $= -10 - \#ofmines$

These rewards were back propagated with a discount rate of 10%.

5 Results

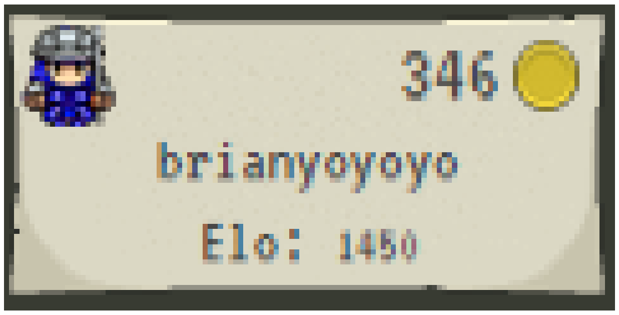

Figure 4: Peak Ranking

Our bot, brianyoyoyo, was run in training mode and the arena for 150+ iterations (see http://vindinium.org/ai/3anzylvh for archive) and updated its JSON file accordingly after each iteration. After each of the iterations a copy of this file was also made so that we could observe any trends in reward values over the simulation.

At first our bot seemed to behave randomly and quite erratically. However, over time it showed improvements and some degree of intelligent decision-making. Over the course of its time in the arena our bots rank improved from a starting value of 1100 up to a peak value of 1450.

By examining the absolute difference in expected reward values between iterations we began to observe a trend and we able to produce the graph in figure 5.

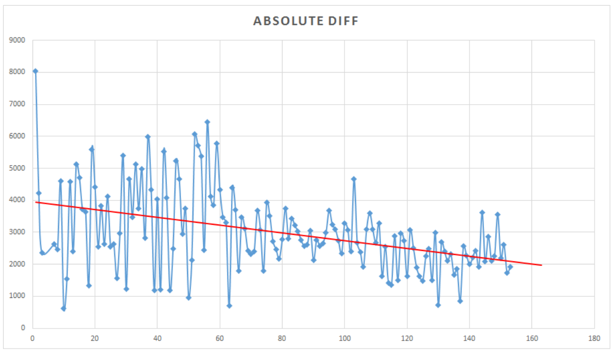

Figure 5: Absolute change in reward estimates

6 Conclusions and Further Work

The graph in Figure 5 shows a downward trend in the absolute change in estimated reward values for each action state pair. Over time we would expect this trend to continue and converge around the correct expected reward values. This indicates that our bot is learning about its environment through each game iteration.

However, in terms of game performance our bot has some shortcomings and often makes poor decisions. At times he appears to be indecisive, wasting moves switching between actions. In part this erratic behaviour could be attributed to his 10% exploration factor where in he chooses a random course of action.

A possible improvement to our current method is to take advantage of the fact that Vindinium is a game of perfect information and use other agent behaviour as training data. The actions of enemy agents in a match could be used as additional training data to better estimate the expected rewards. This approach would also provide a solution to the issue of exploitation over exploration. By using the enemy agents as the training data, our bot is free to fully exploit the most rewarding policy while gaining data from the others.

The rewards system could be modified and experimented with to try and produce a better decision policy. A genetic learning algorithm could be used to modify the rewards given and observe the best outcomes. This would take a lot of training iterations to carry out.

Vindinium also offers the opportunity to develop multi-agent bot architectures where bots can be developed to work together. This could be another interesting area of future work.

7 Breakdown of Work

We collaborated closing for most of this project, however tasks were assigned approximately as follows: Brian :Implementation, Design, Functional Testing. Carl :Research, Debugging, Performance Evaluation and Tweaking.

References

[1] Bcc interview with prof. stephen hawking.

[2] Solving the bellman optimality equations: Basic methods.

[3] Vindinium home page.

[4] Andrew G Barto. *Reinforcement learning: An introduction.* MIT press, 1998.

[5] Johan Hagelbäck and Stefan J Johansson. The rise of potential fields in real time strategy bots. *AIIDE*, 8:42–47, 2008.

[6] Melanie Mitchell. *An introduction to genetic algorithms.* MIT press, 1998.

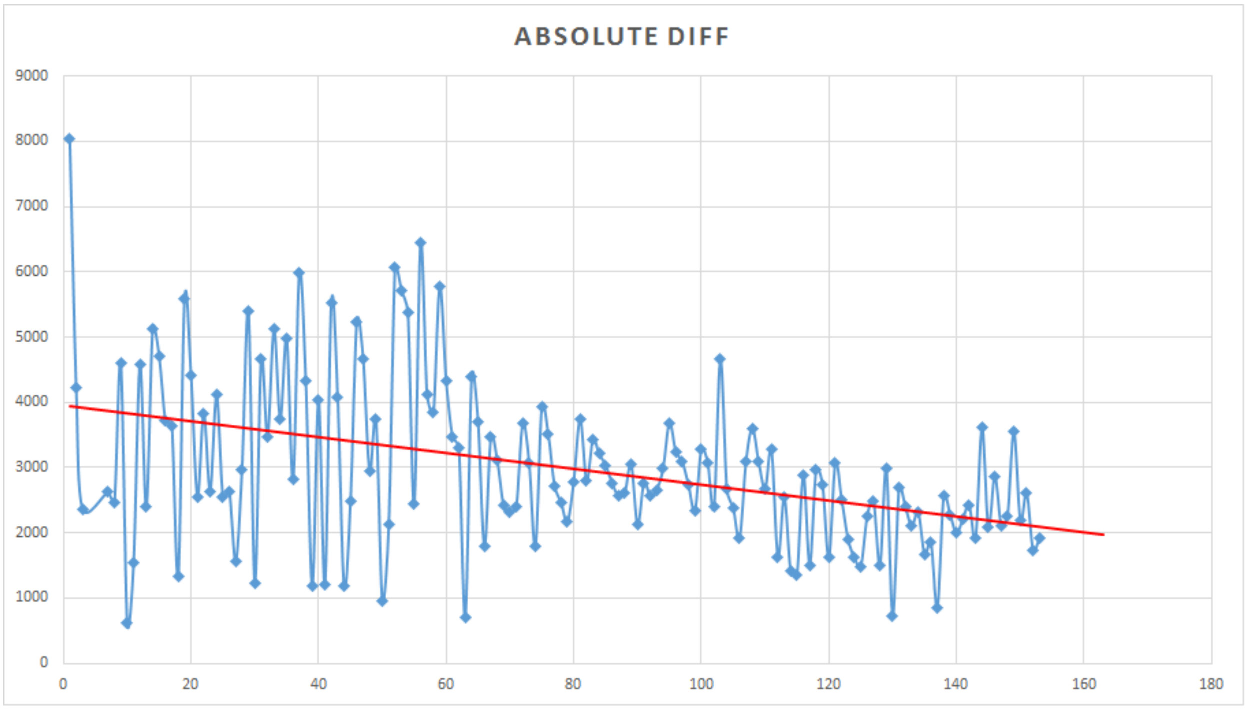

Figure 6: Absolute change in reward estimates