‘Ants’ - an AI Project

Daniel Shriki, Ofer Yehuda, Avner Duchovni.

Problem description

Our project is to create an agent to ‘Ants’[1], a game which was published by Google at 2011 as an AI competition. The game simulates ant colonies that fights for food in order to thrive. Each player controls an ant colony, which consists of one or more nests and one or more starting ants. The game world consists of three elements: land, water, and food. The game is turn based, and in each turn each colony decides it's action - which ants should be moved, and where. After an ant reaches a food item, a new ant will be produced at one of the player’s ant hills, in the following turn.

Aside from collecting food, when ants from rival colonies get too close to one another, they will initiate a battle. The result of the battle is determined by the number of ants under control byeach player in the battle zone.

Why we chose this problem

Ants are amazing. Besides being capable of lifting weights that are many times heavier than their own body weight, they exhibit extraordinary cooperation and communication skills. These features may make colonies of ants sometime appear as a single living organism. The levels of order and sophistication displayed by such an ancient and 'simple' creature, merely a small bug, are of great interest. We therefore decided to give our best in creating our own artificial ant colony.

Our goals

Our goal was to create an agent that will control an ant colony, and make it thrive. The agent should do everything to help the colony prevail, including protecting our nests and ants, eliminating any competing colonies, and gathering food, all in order to increase the colony’s population.

[1] http://ants.aichallenge.org/ - There is a game demonstration in the home page. better than our explanation!

Methods

Reviewing 'post mortems' – reports made by leading contestants in the original **ants-challenge** that took place in 2011, detailing their approach and implementation, gave us a general idea of the methods and techniques already applied to the game. These consisted of reflex agents incorporating classical methods like expectimax and alpha-beta searching. We felt that taking the same route wouldn't be the most educational choice. Furthermore, we noticed the absence of any reinforcement agents - which meant challenge. Considering the recent successes in the field, we we curious whether these techniques can be applied to the ants world just as successfully.

Reinforcement learning algorithms we've seen in class include policy iteration and value iteration. Although these algorithms are guaranteed to converge, they require a model of the environment. Since the world consists of multiple agents (competing ant colonies), these are impossible to produce reliably, and would require us to make strong assumptions regarding the other agents. Under these circumstances, the guarantee of the mentioned methods to converge seizes to apply.

The multi-agent environment is best suited for learning by a model-free algorithm, such as Q-learning. However, Q-learning is based on learning of specific state-action pairs, and most, if not all states should be learned in order to produce a reasonable agent. In particular, Q learning requires that the state space and the action space wouldn't be too big, because otherwise learning becomes unfeasible.

Exponential state space

Every spot in the map can be either land or water. Additionally, in each land location there can be food or an ant of each of the players. This means that for a map size of n tiles there can be many possible different shapes of maps (A map must be rectangular, so the number of different map dimensions is proportional to the number of divisors of n). For each given map dimension there are almost 2^n different possible maps, since each tile can be either land or water (At least two tiles must be land, for the nests of two players). Furthermore, for some maps there can be more than $(number\ of\ players\ +\ 1)^n$ different states, given that on a map that has no water each tile can be occupied by food or by an ant of any one of the players.

Since that not only state space for a single map can be huge, but an agent should be able to operate in different maps, the state space is extremely large. If the number of tiles in the map is denoted by n, we get that the state space is $D^{2^n \cdot (P+1)^n}$, where D denotes the number of divisors of n, and P denotes the number of players participating in the game. This is a super exponential state space, and such a state space cannot be learned by a naive Q-learning algorithm.

We decided to deal with the large state space by using approximate Q-learning, as learnt in class. Using approximate Q-learning we make a transformation from the state space to R^n, where each coordinate is a single feature that receives real values. During the learning process each coordinate of R^n is assigned a weight, signifying its importance. When using this technique, the specific state we consider doesn't matter, since it is characterized by

several function that replace it. This approach allows the generalization of the states, and in particular, allows handling states we haven't encountered while learning.

Exponential action space

While approximate Q-learning allows dealing with large state spaces, even after learning is over there is a problem with working with large action spaces. When considering the agent's next action, the Q-value of each state-action for each valid action must be evaluated. In our problem, the number of actions is exponential in the number of ants we control. When a player has n ants, his action space can reach 5^n actions. This again is a strong exponent, and makes applying even approximate Q-learning directly unfeasible. There was no option other than dwindling the action space, somehow filtering actions in a way that is likely not to eliminate too many good actions.

Breaking down action space

In order to reduce the action space, we changed the way actions are looked at. Instead of each action being a function from the player's ants location to the directions each ant will go to, (which means there are indeed 5^n possible functions since each an ant can stay in it's place), we looked at an action as a finite series of function, where each function represents a single ant. This way for each ant there are 5 different possible functions, and the action space consists of choosing a function of out 5 possible ones, once per each ant. The result is an action space of size $(\#number\ of\ ants) \cdot 5$.

Reducing the state space from 5^n action to 5*n states means not considering most of the possible actions for a given state. However, it was critical in order to allow an efficient agent. When not evaluating each action there is a high possibility that the best actions may not even be considered. However, considering the complexity of the problem there seemed to be no other resort. More information regarding the limiting of the action space and other considerations revolving it can be found in later in this paper, under 'Further Research'.

Ants symmetry

Breaking down each colony-action into a series of ant-actions meant that features and rewards would have to be personal, meaning that the features used to describe a state should represent a single ant's role and position in that state. Accordingly, each ant should be rewarded by its own contribution. However, since all ants are the same, and all ants choose actions according to the same evaluation method, the success of the whole colony is a good indication that the actions taken by the ants was good, and should be rewarded well.

Using 'global' features and reward could result in a situation where most ants' actions were good ones, and a high reward is given to each ant. Such positive reward can still be given to ants which took a bad action, which will, in turn, hinder the learning process. We concluded that global rewards can still be a useful tool, although in general it seemed a more appropriate way to evaluate and reward each ant by its own contribution.

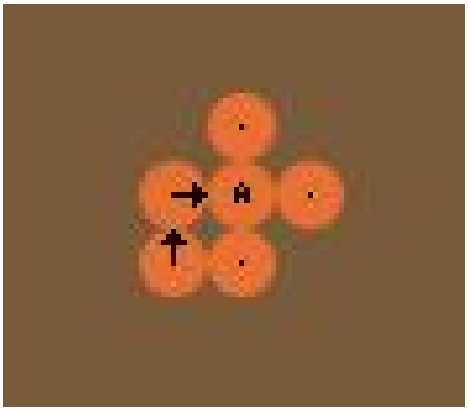
Communication between ants

Choosing actions at the ant level instead of the colony level gave rise to another difficulty - the need in communication between ants. Before breaking the action space into ants-actions, we were 'overseeing' the ants. With each action evaluated, we automatically

took into consideration the ants' effect on one another. However, when each ant chooses its own action, one by one, actions taken by previous ants isn't immediately taken into consideration. This could result in an ant choosing to go to a location that will be already occupied by another ant (which, by the rules of the game, results in both of the ants' death). To avoid such cases, we had to implement a way for the ants to be aware of each other's actions.

Death is inevitable - Anthony's sad case

The sequential manner of choosing ant actions brought more difficulties than expected. Even when ants were made aware of their counterpart's actions, some undesired and even morbid side-effects occurred. To demonstrate such a side-effect, we will describe the case of a brave ant named Anthony.



"Anthony's choice"

Imagine brave Anthony standing, ready to go and help his colony thrive. Next to him are his five friends. The ants that are above, to the right, and below Anthony get to choose their action first, and they choose not to move at all. Next, the ant on the left gets to move, and chooses to go right, to Anthony's own location. Then, the ant on the bottom-left gets to move, and chooses to go up. Now it's Anty's turn to move. However, anywhere he'll decide to go he will bump into another ant, which will result in his and one of his friend's untimely death.

The only way to avoid this kind of situation, where an innocent ant dies because of friendly ants' actions, is to forbid ants from going to places where other ants are present. This is very problematic, because this doesn't allow, for example, ants to go in a single file (Like we so often see ants do in nature. Limiting such behaviour is inhumane and we refuse to even consider it).

Rewards

When designing the rewards, we were concerned of the tradeoff between having minimal rewards which represent the purpose of the agent more truthfully, yet slow down learning, and between adding additional rewards. Adding additional rewards for what we see as positive behavior could accelerate learning, but with the risk of introducing a bias favoring certain policies.

The score of the game is calculated by the number of enemy ant hills a player eliminated, minus the number of hills he lost during the game. These introduce very little reward for very long sequences of actions. Learning in this manner would require higher computing abilities and more time than what we had available.

We added additional rewards in order to accelerate learning:

- Collecting food: each ant that collected food in the previous turn is awarded 5 points for each collected food item.
- Map exploration: each food item discovered by our ants, which was not visible last turn, grants 1 point.
- Killing an enemy ant: for each dead enemy ant in the attack range of ant agent, it is awarded 50 points.
- Dying: an ant that has died in the previous turn receives -10 points.
- Stepping on enemy hill awards the ant with 2000 points (a bit over the top but it was a rare event)

Notice that these rewards combine both global rewards (rewards that are equal for all ants), as in positive and negative reward at the end of the game in case of winning or losing, along with personal rewards (reward which is not equal for all ants), as in an ant receiving a reward when it has eaten food.

Furthermore, rewards can collide in many cases. For examples, when two ants from rival colonies get too close to each other, they initiate a battle. If they are the only ants in the battle radius, they will both perish. This will result in both a positive reward, associated with killing an enemy ant, along with a negative reward, associated with dying. Similarly, killing an ant may involve missing the opportunity to eat a food item, thus involves missing out on one reward for the benefit of another.

Other rewards seemed to produce peculiar behavior in certain situation. When giving too high of a reward for finding new food items, we have witnessed situation where ants will find food, then will go in the opposite direction, only to re-discover the same food item over and over again, all instead of eating it.

All in all rewards seemed to be a bit of a guessing game, and in more than one case we had hard time giving a reasonable explanation of the change of feature weights assigned by the algorithm after altering the rewards we use and their size.

Features

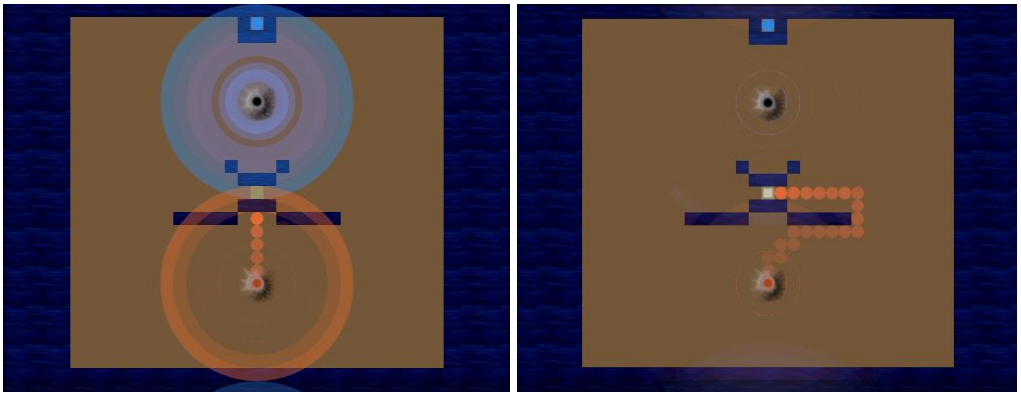
Designing of features was focused on two elements we considered most important in growing a healthy ant population: exploring the world and eating as much as possible.

Food collection

Feature 1: Distance from closest food

Given a ***state*** and ***action*** for ant ***a***, this feature calculates the distance of the closest visible food to ***a***'s location after executing ***action***.

At first, the distance was calculated using Manhattan distance. This worked well most of the time, except when the manhattan path was obstructed by water. We implemented A* search[2] for distance calculation using the manhattan distance as a heuristic. This fixed pathfinding issues but cost us in running time.



ant behavior when using manhattan distance (left) and A (right)*

Feature 2: Will eat food in next turn

This feature is inspired by the Pacman assignment in Assignment 4. It equals the amount of food consumed by agent given ***state*** and ***action***.

Feature 3: There is a closer ant

The previous features allowed our ants to see that eating food was desirable. One behavior that we noticed though was groups of ants that would all go to the same food, even though only one was actually needed in order to eat it.

[2] Code taken from aima-python github repository: https://github.com/aimacode/aima-python

Our first attempt to mitigate this behavior was to add a new boolean feature, which would indicate whether the ant was the closest one to the food. This feature was notorious for introducing unexpected behaviour, so it was mostly deactivated.

Exploration

Another feature was designed to encourage our ants to go and explore the map. This behavior is critical, since in order to grow the population of our hive we have to eat food, and in order to eat food we must first find food to be eaten. Thus the feature we added was the percentage of the map visible by the sum of the colony's ants.

These features and reward do collide, just like rewards. One encourages eating food, and the other encourages *seeing* food, which means that it discourages eating food. We have made quite a few attempts with this behavior, and encountered some unexpected results. For example, the higher the reward for gathering food was, the smaller weight of the percentage of the map that is visible received. After investigating we've come to the conclusion that this is explained by the fact that before eating a food item and getting a high reward, many ants tend to cluster around this food item in order to eat it. As a result, there was an association between vision of a smaller percentage of the map with a high reward.

Other features

The following features had minimal/negative effect on the behavior of the agents.

Enemy ant hills

We had two features for enemy ant hills identical to the food collection features. They didn’t receive the expected weight in training, perhaps because it was extremely rare for our ants to manage to destroy an enemy ant hill.

Ants in attack range

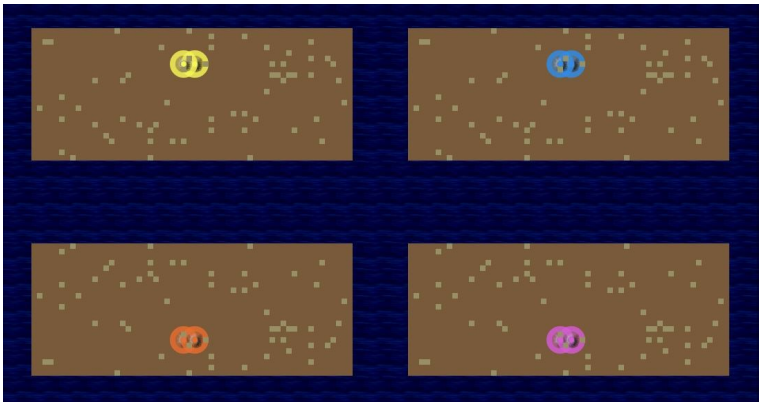
We had a feature for number of enemies in attack range and number of allies in attack range. The idea was for the agent to use it for battle, as they are determined by these numbers. Strangely, it introduced in ants the desire to huddle together too much, and so we deactivated it.

Evaluating our agent

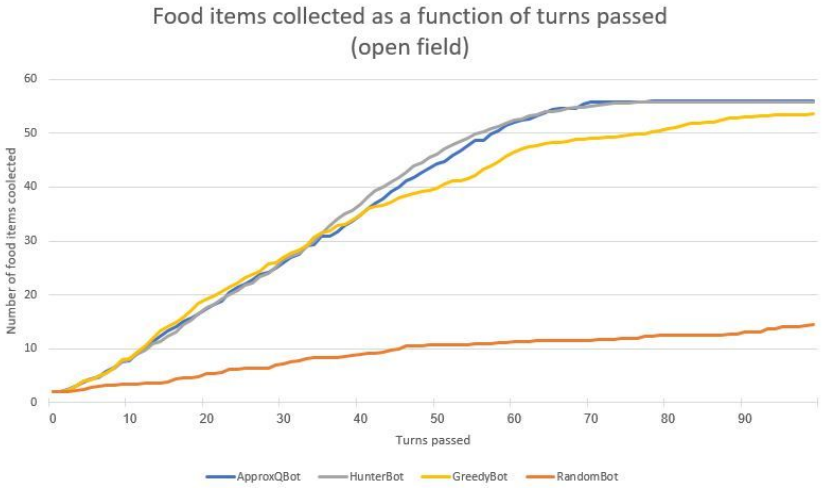
Food Gathering tests

Open field food gathering

In this test we create a big map, and let 15% of the land of the map be covered with food items which are placed at random. After learning we ran the game 5 times, letting the game run for 100 turns each game. Then we took the average of the 5 games, and compared the food collection rate (cumulative) of our bot against 3 different types of bots - HunterBot, GreedyBot, and RandomBot. Each of these bots follow different heuristics.



Map for comparing food collection rates on an open field

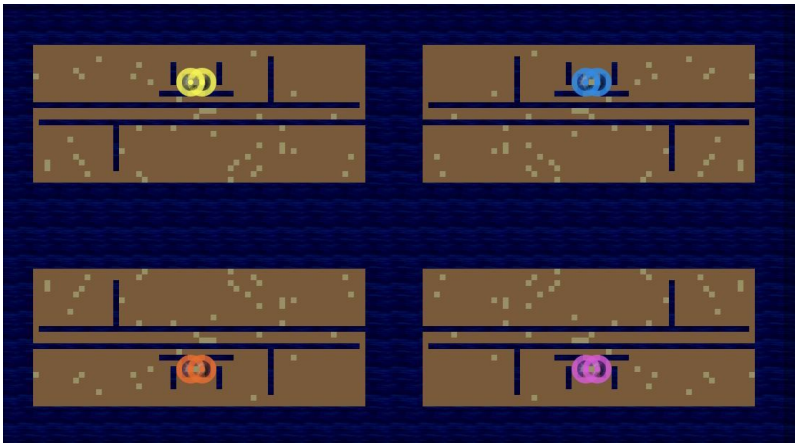


Food collection rate of different agents in an open field

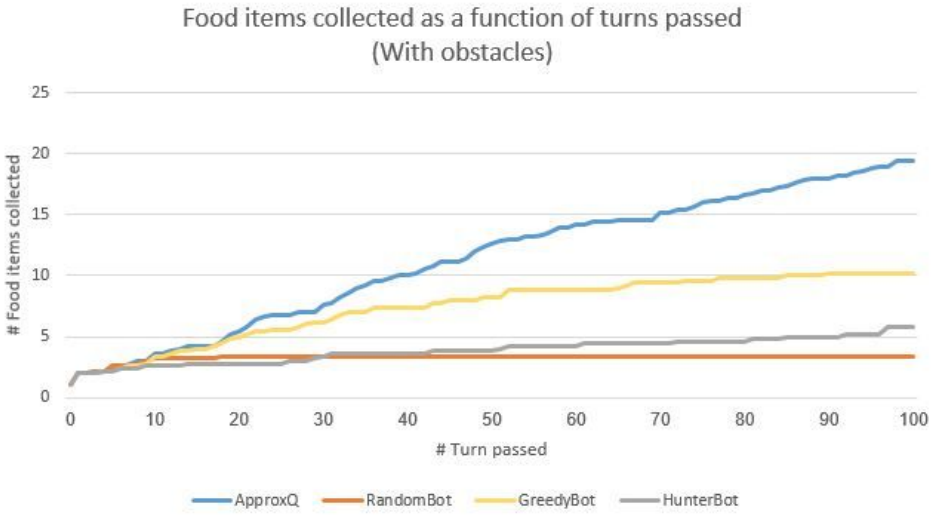
It seems that our agent has a slightly better performance than the greedy bot, but is behind the hunter bot but only by a very slight margin.

Food gathering with obstacles

This test was made exactly the same as the previous one, except that this time we built a map that has many obstacles.



Map for comparing food collection rates on a map with obstacles

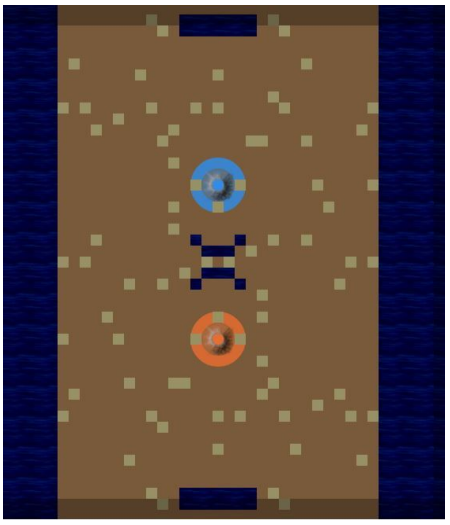


Food collection rate of different agents in a map with obstacles

It is clear to see the benefit of pathfinding - our agent outperforms all other agents by quiet a lot. In this experiment our agent didn't finish collecting all food items on map, since it was bounded for 100 turns. It looks promising that given more time the agent's performance will exceed it's competitor by much more that is represented here.

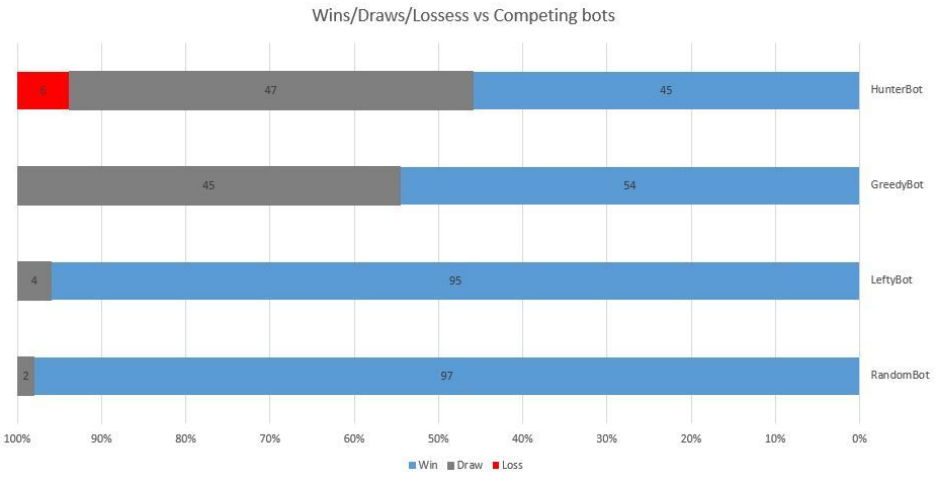
Facing enemy bots 1-vs-1

We let our bot compete in a player vs player scenario vs four different bots: HunterBot, GreedyBot, LeftyBot and RandomBot. We tested our bot on a standard map, with some food items and a couple of barriers.



Map for testing our agent in 1-on-1 battles

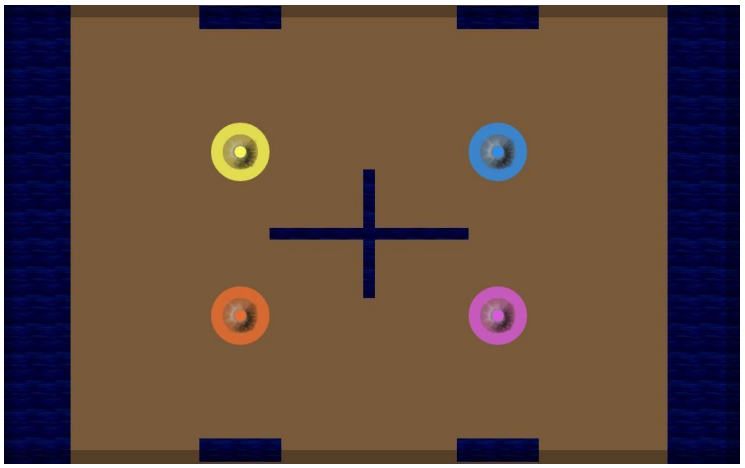
Out of 100 games player vs each bot, our agent presented good results against each of the bots, almost never losing, and winning many times.



Statistics of 1-on-1 battles against competing bots

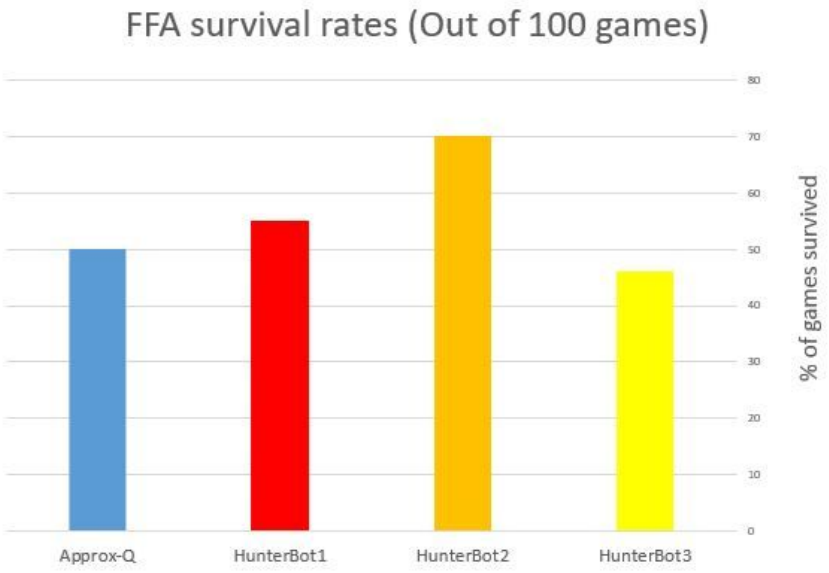
Free For All

An even harder test is the FFA format, where everyone fights everyone. We let our agent face three copies of the best default bot, the HunterBot. We ran 100 games, each consisting of 1000 turns, and plotted the percentage of games each of the agents was able to survive.



Map for testing our agent in FFA battles

The results of the FFA test were less uplifting - two of the three HuntBot agents were able to survive more games than the approximate Q agent. It was intriguing that our approximate Q agent was very successful against HunterBot when facing him 1-vs-1, but less so in the FFA scenario. We could not come with a satisfactory explanation for this situation.



Statistics of the FFA battles against three HunterBots

Further research

Limiting the action space

It seemed reasonable to limit the action space by looking at a single ant and evaluating its action. In our implementation the ants order is chosen at random. Changing the order of ants for which an action is chosen could possibly improve our results. For example, we could evaluate each ant's actions, and then allow the ant with the highest rated action to choose it's action first.

However, in this manner of action choice, after allowing the first ant to move, other ants moves which we have already evaluated may no longer be valid. This will have to be resolved in some way, most likely by re-evaluating those ant's actions, or by keeping in memory all ant's action evaluation. Re evaluating ants' actions can theoretically increase the action space and make it exponential in the number of ants again. On the other hand, remembering each ant's actions evaluation will take up vast amounts of memory.

No matter which solution will be implemented, if the exponential action space will be reduced to a linear action space will result in most of the possible actions to be overlooked. We could not come up with an idea how to assure that this kind of reduction doesn't omit the best possible actions, at least for some of the possible states.

Computational cost of path finding

As the number of ants increases, using A* for path finding becomes impractical, as there is a time limit on a bot's move. There are several alternatives to consider:

- Pre-Calculate distance between each pair of tiles in the map using BFS. This option is viable if the time allotted in the beginning of the game for the bots to initialize is long, or can be done slowly over the course of the game.
- A form of depth limited A* search. It might still give us a better estimate than manhattan.

Non Linear features

In some cases we were thinking about breaking the linearity of features. It seemed that some features have high importance when they get higher values, but not necessarily in a linear fashion. For example, the proximity of an ant to its nearest food is of great importance when the distance is small, since with little effort a good gain to the colony could be achieved. However, when ants were further from food items, it seemed that in most cases they shouldn't presuit getting to that food, rather it may be more beneficial for them to go and do something else, like explore. It seemed that using a sigmoid relationship between the distance and the feature value could yield better results. However for the sake of simplicity we hadn't incorporated this in our final bot.

It also seemed a viable option to combine both linear and non-linear features describing the same trait of the game. For example, allowing both a linear and a sigmoid features of the

minimal distance of an ant from a food item. This way we allow the agent to choose which of these features to give a higher weight, and when trained for a more specific goal, and not for the most general case, it will have more flexibility in changing it's learnt policies and weights.

Battles

During our work we tried to introduce features that deal with battles, but without much success. Perhaps there are features that can cause the agent to do battle well. Alternatively, one can exclude ants in danger (with enemies in range) from being handled by the learner, and instead use some form of minimax tree to decide the best action for those ants. Perhaps battles are just too complex for linear functions to be enough.

Neural Networks

Obviously, we were curious to see if any of the recent advancement in Deep Reinforcement Learning could be applied to the problem. We had attempted to implement an agent similar to our approximate-Q learner except using a neural network inspired by this[3] blog post. There is an implementation located in our_stuff/NeuAgent. We haven't managed to achieve competent behavior, but it still seems like a promising direction.

[3] http://karpathy.github.io/2016/05/31/rl/

Final thoughts

This was the first project we had to do for University. The difference of such type of work from standard assignments in different courses came to us as a surprise. Unlike any other assignment so far, in this project we had to take care of everything. Starting with picking a problem, through deciding on our own goals, adjusting the platform, coding, testing and so on. A big part of our effort was dedicated to technical work and handling bugs which were completely unrelated to the material and techniques applied on the problem at hand.

When technical work was over, and we were left with the actual learning algorithms we wanted to apply, we had to face with the process of designing features and rewards. During this process there were no clear 'right answers', just heuristic experimentation, and that was very challenging. It was hard to tell when a problem was caused by the linear nature of the Q-function approximation, and when it was simply due to lack of good ideas.

It is clearer now why the leading bots in the 2011 competition all used mainly heuristic methods. Classical reinforcement learning techniques don't handle the challenges we presented during this report as competitively as the heuristic methods. Nevertheless, there is a certain satisfaction one gets seeing his bot choose strong moves without direct coding.

Even though we haven't been able to successfully use any "Deep Reinforcement Learning", seeing how StarCraft 2 is likely to be conquered by the AI community in the coming months, we believe a bot using those techniques will probably beat the winners of the *Ants* competition easily. We hope to continue our work on the game even past this project.

Appendix A - Adjusting the platform: 'through the ant hole'

Modifying the game to support learning

The AI ants challenge took place in 2011. At that time, neural networks and reinforcement learning weren't at the forefront of AI research, so the challenge wasn't designed to support it. On the other hand, it was designed to support multiple programming languages. Therefore, the game server receives the commands to run the bots, and for each bot opened a new thread with it's own input and output streams. After each round, each bot's thread is restarted, even before the bot terminates. Obviously, this posed a difficulty for learning, as we needed continuity between game rounds.

To overcome this obstacle, we modified the game's code to support one special bot, the learner, which had the same thread running throughout all the rounds.

Identifying ants between turns

At each turn the bot receives from the server the new state of the map; where there is food, ant hills, ants and dead ants. Since we chose to have our reinforcement learner learn a policy for a single ant, we had to figure out how to identify ants between turns, so we could assign the right reward for the right (state,action) pair. The difficulties were understanding when an ant has died, and when an ant is blocked, and therefore not where we expected it to be.