Bomberman

An AI Course Final Project

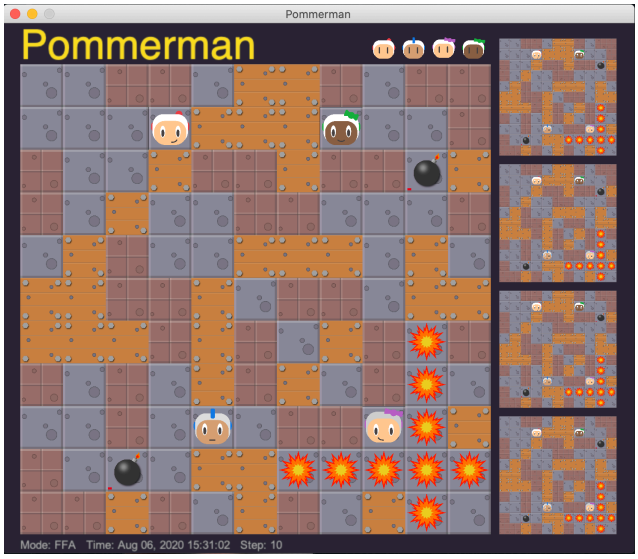
Oded Fogel Tomer Lankri Arbel Amir Or Nokrean

Abstract

We attempt to solve the bomberman game using Approximate Q-Learning and Adversarial Game Trees. We managed to find evaluation heuristics that resulted in a very efficient game tree search agent, and found very impressive results with Approximate-Q Learner. this resulted in very good results, especially in comparison with similar past project.

Introduction

Bomberman is a strategic, maze-based video game franchise originally developed by Hudson Soft. it's a 2-D real time strategy game, where the players need to place bombs that can kill enemies and destroy "wooden walls", while avoiding getting hit by any of the bombs. The bombs have a constant strength which varies between 1-4 , where strength of 1 will emit flames in the place of the bomb and 1 flame for each direction (up, down, left, right), while strength of 4 will shoot up 4 flames in each direction.
As we searched the for a python compatible version of bomberman that will fit our needs, we found the Pommerman open source project [2] which made exactly for this purpose – creating AI agents and competing them online in the Bomberman game. It looks like this:



We used this platform to create several AI agents and pit them against each other, while improving their heuristics and algorithms, comparing their performance and even trying to beat them ourselves (unsuccessfully). An important change we made from the original game, is that we removed the power-ups functionality, since the simplified version of the game with no power-ups seemed challenging enough.

One of the big challenges in this game is that the time between placing a bomb and it's explosion is long enough for the board to change rather drastically, causing a lot of environment changes. This setback will come into play when we're on a multi agent game and we're not sure how each of the agents will react to each state. Even if we try to guess and evaluate their moves, accurately predicting their moves is impossible. In order to understand and evaluate the implication of our specific bomb damage, we will have to go deep down in the game tree, which is not so gameplay suitable. This is why we needed to find evaluation heuristics that could evaluate the ramifications of the current state. This problem popped up also in our Q-Learner agent training, when the agent had a hard time figuring out the connection between an action and it's outcome.

That's why we chose two different approaches: the Alpha-Beta pruning for adversarial game tree and Approximate Q-Learning built on features vector of our own design, and modification of the reward calculation.

Though most of our algorithms are made for 1 vs 1 games, we support gameplay against one, two or three players! This make the game more interesting and our problem more complex, which makes everything a bit more fun.

As part of this pommerman package, we got and agent called Simple Agent, which acts by this order:

- Moves if he is in a safe place
- Maybe placing a bomb if near enemy
- Move towards enemy if there is one in exactly 3 reachable spaces
- Maybe placing a bomb if near wall
- Moves towards a wooden wall if there is one within 2 reachable spaces
- Go to a random valid direction.

We used this Agent, and better version called smartRandomAgent [3] that uses ActionPruning to randomly select an action from a smaller actions-pool, to measure our agent's performances.

In order to measure our agents we pitted them against the simple and smartRandom agents, and against each other.

Approach and Method

Min-Max agent (with/without Alpha-Beta Pruning)

Min-Max: At first, we've implemented Min-Max agent and measured it's performance – we saw that for a 1v1 game, any depth above 2 on a is not computable and reasonable to play the game with. The general idea was to simulate the game in a way that we will be the max agent, and all the other (1-3) player(s) will be combined together and be the min agent. It can be seen as if all the enemies are controlled by the a single enemy player, when its top priority is to kill out agent. Although this is a simplifying assumption, we preferred to look at the game this is the potential worst-case behavior from our agent's point of view.

The problem with adversarial game tree for more than one enemy, is that the branching factor is huge: it's 6 for the max agent and 6^3 for the min agent(s). That made our agent runtime very slow for any depth greater than 2. So we considered working with the Alpha-Beta pruning in order to reduce runtime.

Min-Max with Alpha-Beta: After applying Alpha-Beta pruning on our Min-Max tree, the results improved a bit, but we saw that we need to modify our branching factor in order for it to run more smoothly. We had to figure out which of the states and actions are problematic or useless - we saw that the agent explores actions and states that are irrelevant such as walking right to his own death (placing a bomb in a way that blocked any possibility of running from the blast – in a kind of death-pit - figure 1).

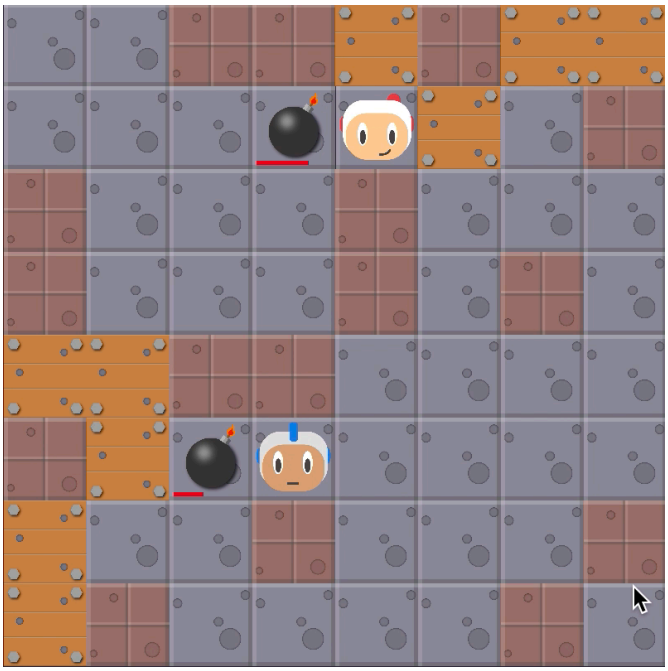


Fig. 1. The top player got himself into a pit. It's all over for him.

It's apparent that such branch isn't worth exploring, but the problem was that the bomb time to explode is 9 steps, so the agent had to explore quite a bit before realizing that the branch is a bad one. We chose to eliminate some of the possible actions from the action-pool used to expand the trees, if said actions caused the agent to walk right into it's death, otherwise pruning the tree down from situations such as described above.

a) Evaluation heuristics: Because of the big distance (in the tree) between the bomb-laying action and the actual result of it (the explosion), we needed to create an evaluation heuristic that allows the agent to "predict" the future value of a current state. For that reason, we used features like bomb-locations (that determines the future blast) and freedom of movement (that allows escape from the blast).

We began noticing different results and gameplays by choosing different evaluation functions, and their weights. We started off some basic weighting vector and features, such as how many bombs are on the board right now, how many agents left, how many breakable walls our agent destroyed or the distance from our enemies and similar others. Then we added features that allowed the agent to avoid committing suicide, such as checking if the agent is on flame (that marks the explosion area) or near them - and the same for bombs or enemies. Then we became offensive - we evaluated bombs near our enemies - that will cause their death down that branch, and clearing our way to other agents and so on.

The features we ended using are:

- am_i_dead: This feature indicates if the agent given is currently dead, i.e. out of this game.
- distance_from_enemy: This feature indicates the distance from the nearest enemy on the board.
- is_near_bomb: This feature indicates the situation of our agent being in bomb explosion range, which will lead too his death.
- is_on_bomb: This feature indicates the situation of our agent being in bomb position, i.e. also in the explosion range, which will lead too his death.
- is_on_flame: This feature indicates the situation of our agent being in flame position, i.e. also in the explosion range, which will lead too his death.
- is_stuck: This features indicates the situation of the agent getting into pit, which will lead to a certain death.
- is_there_a_bomb_next_to_wall: This feature is for our evaluation of the agent activity in the game. we want to make him be an active player in the game, so we encourage him to put bombs next to breakable walls in order to demolish them.
- num_bombs: The number of bombs on the board.
- number_of_broken_wall: The number of walls broken by the steps we consider taking.
- smallest_dist_from_bomb: The distance from the closest bomb.
- num_enem_actions: The number of the enemy's possible actions in this state. It was used in order to

take advantage of the situations which the enemy get into corners, and act accordingly. It helps us prioritize situations where the enemy is in corners, pits and etc. (all of them are states where it is easier to kill him).

After we got some good results with those features (weighted to produce the "balanced agent" described below), we created different strategies by changing the weights to produce the desired behaviors. We ended up with the following strategies:

- Balanced: This was the best evaluation we created - takes all of the features and emits the smartest action to do for the current state. "Knows" to be aggressive and offensive, and as his name suggests, balancing between the two.
- Aggressive: He will prefer moving closer to the enemies and placing bombs near them / near breakable walls, in order to almost always make a difference in the game. May die in process. We gave higher rates to placing bombs near enemies/walls. As we wanted to create an aggressive player, we changed our feature weight in such a way that it gives bigger bonus on laying bombs, and specifically bombs that are closer to enemies.
- Pacifist: Will avoid fights, will run away from bombs and try to attack when feeling safe. As we wanted to create a pacifist player, we changed our feature weight in such a way that it gives punishment for being near an enemy or bombs/flames, ad in general, punishing for laying bombs.

b) Results: In the end, we were able to use the tree with a depth of 4 in the 1vs1 mode, in a 3-players-game a depth of 3 was still practical (albeit slow). Unfortunately, in a 4-players-game the maximum "real-time" depth was only 1 (and even then, the game was very slow). All of those configurations resulted in a slow game, but when reducing the depth by 1 for each, the result was real-time speed.

All of that resulted in a very resilient player - none of us could ever win against him, the "built in" AI agent we got with pommerman package loses repeatedly, and even a better agent – "SmartRandom" – lost to our agent on 100% of games!

After we got very good results with a "balanced" approach for the heuristic, we started running it against our own Aggressive agent and Pacifist agent, and got this results:

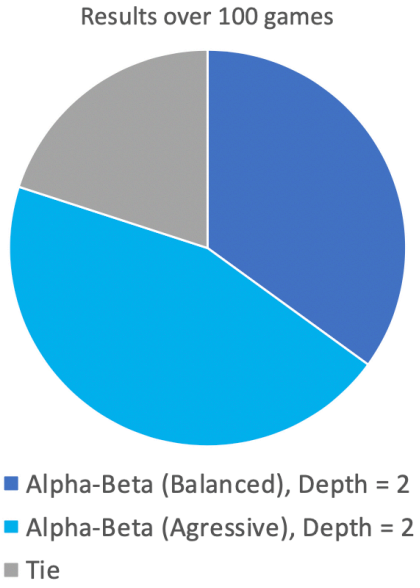


Fig. 2. Balanced (d=2) vs Aggressive (d=2)

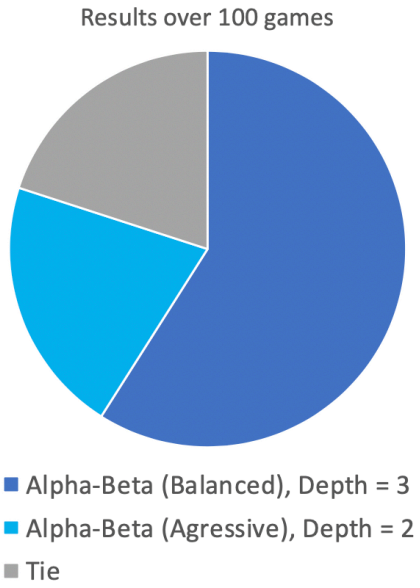


Fig. 3. Balanced (d=3) vs Aggressive (d=2)

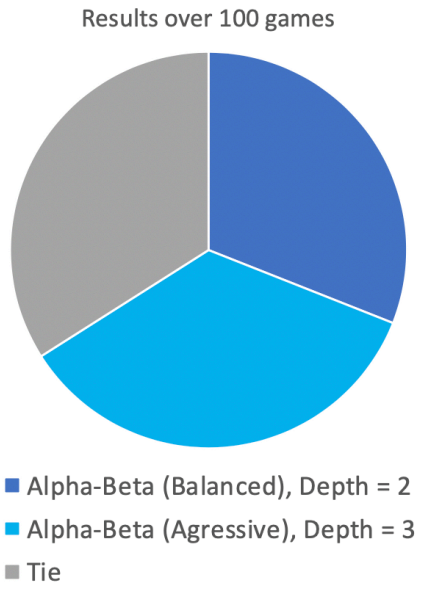


Fig. 4. Balanced (d=2) vs Aggressive (d=3)

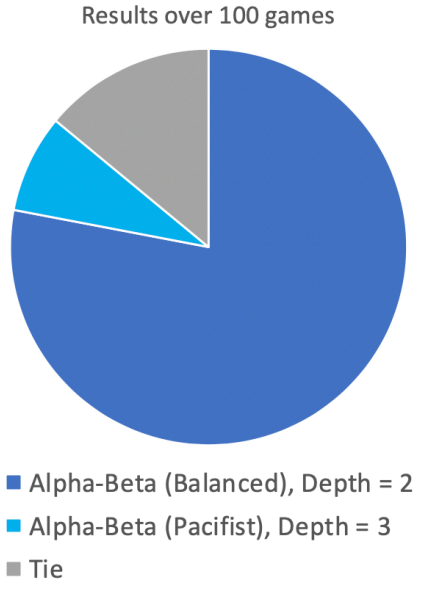


Fig. 5. Balanced (d=2) vs Pacifist (d=3)

Approximate Q-Learning agent (AQL)

We implemented and working with a Q-Learning agent, which failed to work because of the complexity of the board and the game states, the different and random game boards and the different adversarial approaches. Because of the large state-space of the game, trying to match each game state with a score was very hard to learn. This drove us toward creating an Approximate Q-Learning algorithm, which showed better results.

As can be seen in the results below, the AQL gave us very impressive results.

c) Reward: Because the default reward of the game is given only when an agent is dead (otherwise it's 0), which meant that during most the game itself the weights were not updated, we initially tried assigning our own custom reward to a given game state. That reward was based on more parameters than just the dead\alive status of the agent and enemies, such as the number of bombs on the board and distance from enemy (to increase aggressive behavior) or the number of actions available to the agent. However, with this reward we saw that the agent acts as though as if it was given a heuristic (like a reflex agent), and there was almost no learning process.

In the end, we calculated the reward based only on the game's results – who won and who lost, though we gave different weight to each result:

$$R(s) = \begin{cases} -1000 & failure \\ 100 & success \\ 0 & else \end{cases}$$

When failure means our agent is dead and success means our agent is alive and enemies are dead.

Features: In the process of creating the Approximate-Q Learner, we developed several feature functions which will helps us determine our next action.

First, we used functions like:

- Bias (=1)
- Number of bombs on the map
- Number of walls broken in last turn
- The distance from nearest enemy in relation to board size
- Enemy's distance from the bomb nearest to him in relation to board size
- Did I place a bomb right near an enemy?

And some "death control" features such as:

- Am I in bomb range
- Am I on flame (bomb exploded and I was in range)

We noticed that some of those features (like the number of bombs on board or the number of broken walls) added bias to our learning process, which gave us learning graph that diverges even more than without them. Our assumption was that even though the number of bombs on the board (or the number of broken walls) depends on our agent, it also depends on the other agents in the game, and our agent has no control over them, so we shouldn't reward him for that.

We decided to use only features which our agent has control over, which left us with the following feature list:

After removing the "confusing" features, we're left with the features of the distance from nearest enemy and the "death control" features. We wanted to sharpen our features some more, so we added a feature of "is bombing enemy", meaning - given that our agent is 1 step (by manhattan distance) from the enemy, we check if his

action is placing a bomb. In other words - did our agent placed a bomb right near an enemy (1 slot up, down, right or left of his position). We also added a feature function of bias, which is always one, in order to create some uniform behavior. This left us with the following final features list:

- Bias (=1)
- The distance from nearest enemy in relation to board size
- Am I in bomb range
- Am I on flame (bomb exploded and I was in range)
- Did I place a bomb right near an enemy?

All of those and the reward function $R(s)$ we showed earlier satisfied our goals with the Approximate Q-Learner.

Finding a Converging Learning Function : We examined our AQL agent learning behavior vs SA, to develop a reasonable learning process, before examining it versus a more complicated agent (like our Min-Max).

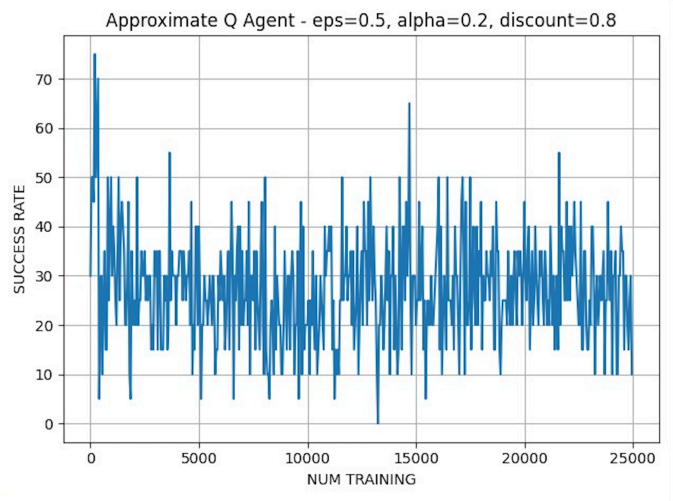


Fig. 6. Here we see the learning progression for an AQL with learning rate of 0.2. We clearly see that this learning progression diverges, with no dependency on the number of trainings.

We tried to fix this problem and reaching some kind of convergence of the learning rate function, so we viewed so previous works [4], and according to them we tried to resolve out problem by lowering the learning rate (α) to 0.05 and got something that firstly looked like good promising results for about 2000 episodes, as can be seen in figure 3:

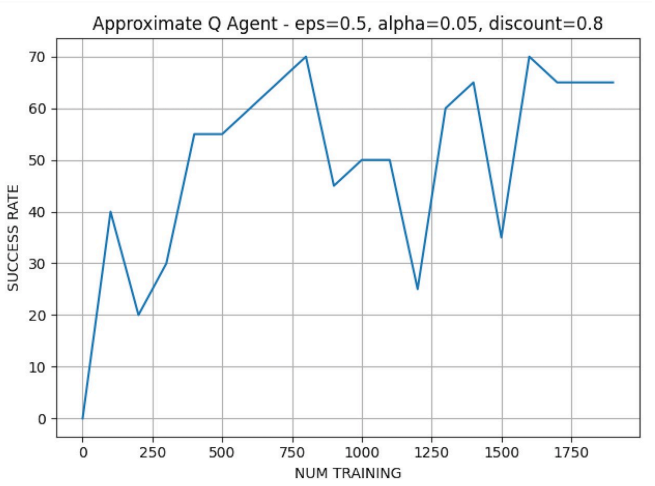


Fig. 7. Training for 2000 with $\alpha = 0.05$

So we decided to let the agent keep on training, to convince ourselves we got a monotonically increasing learning function, so we tried with 8000 episodes, but then we saw that was not exactly the case. The results are presented in figure 4:

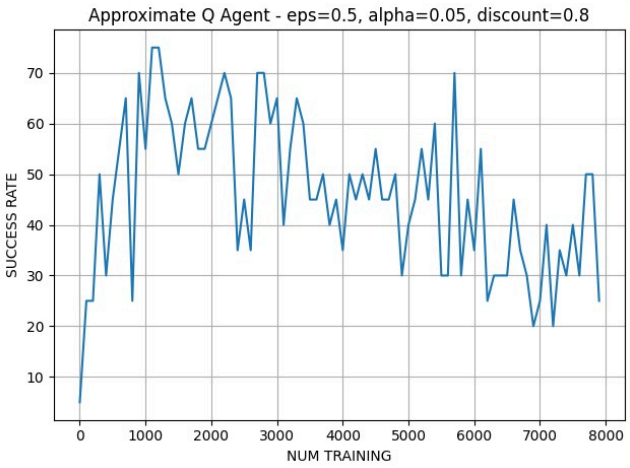


Fig. 8.

However, when we reduced the alpha even more (to 0.02), we did manage to get a convergence when trained long enough, as we see in figure 9:

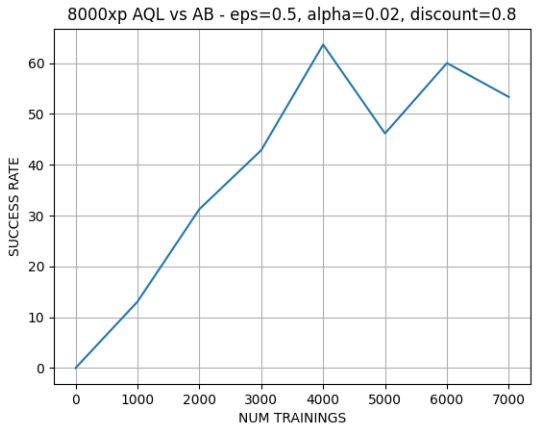


Fig. 9. A converging training with good results.

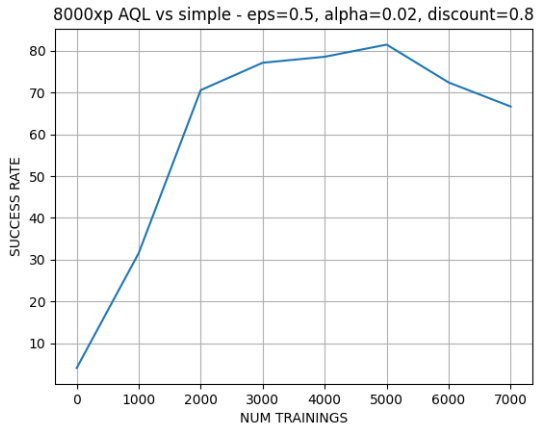


Fig. 10. AQL trained against AB agent of depth 2, tested againt a simpleAgent.

We may have reached even better results if we we're to implement principles from the Machine Learning world, such as: decaying exploration probability and learning rate, regularization loss function, randomizing the initial weights etc. Those were not part of our course so we chose to leave them out of our solution.

Another interesting thing to note is that after training against

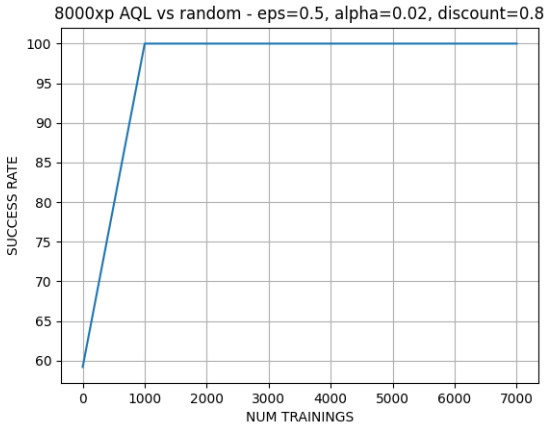


Fig. 11. AQL trained against AB agent of depth 2, tested againt an agent that chooses actions randomly. As can be seen, the agent started winning all games pretty quickly, which means the AQL managed to learn an efficient strategy that doesn't get it killed for arbitrary reasons.

Results

As can be seen above, the Alpha-Beta Balanced agent managed to yield pretty good results, and wins most of the games played against standard agents/players. On the other hand, when we matched him against the AQL, since the AQL trained with the Balanced agent, it succeeded to win most games between them.

As we saw in figures 2-5, the balanced agent was the best strategy for the Alpha-Beta Learner. We saw him triumph even when the depths were uneven, or getting tied up with his opponent. It's worth to mention that the aggressive agent work pretty well as well, and did so also when he was competed with the Pacifist agent.

As we saw above, the AQL managed to win all of the simple agents very easily in most of the games, and when he was trained against the Alpha-Beta agent, he managed to win a decent amount of time (over 60% almost every time). When running the trained agent against simpler agents such as simpleAgent, we got even better results as can be seen in figure 10. This shows that the AQL didn't over-fit to learn a strategy that beats only the type of agent it trained against, but learned a general strategy useful for any type of enemy. This also means we managed to create an objectively good agent (since AB was already very good).

However, we also couldn't get a convergence with the AQL when trained in 3-players and 4-players games. We suggest this is due to the fact that those types of games are more coplicated, and so to efficiently train against them, a longer trainining is required (which we didn't have the computing resources to do).

Both Alpha-Beta and the AQL agents won each and every game against some very smart human players (meaning us). So we can formally state that those are objectively smart and all around good players. Even when we tried to be passive and safe players, we never managed to get away and never won even a single game against the AB agents!

Compared to the 2012 project

This game was already discussed in a previous project from 2012. We had a strong incentive to try and add to the previous results, because we had high hopes for our object performances compared to 2012.

As can be seen, in above results, we were right. Let us explain:

First, we got better final results in comparison (winning %), we managed to go deeper in the game trees (depth of 3-4) and still maintain a playable agent while doing so (vs human, you can play with those agent in real time and they response relatively quickly).

We also got our Q-Learner to work really good, which they didn't, so our work was not in vein.

Further more, we've also added a very important part to our solutions, which is the multiplayer game mode (3 and 4) and got pretty good results there too,

with alpha-beta depth of 2 in real time.

Conclusions

As we can see in figure figure 9, our AQL agent beats the Alpha-Beta most of the time, after the AQL agent was trained against the Alpha-Beta agent. Using the general features the AQL agent uses, it manages to handle very different board types and game situations, which suggests it might be able to deal with a variety of different agents.

For future work, one could train the AQL with many agents and by that developing a more elaborate and sophisticated strategy, that could handle every almost opponent out there, instead of just the one. It could also be interesting to see how an AQL that was trained with small number of agents, will generalize his strategy versus a more variant set of agents - as we had somewhat limited amount of agents to work and train with.

While researching for our projects, we've encountered many ideas to solve this problem, and one that came up often is the idea of the use of neural networks along with the Q-Learning agent, as many attempted to do successfully [6]–[8].

Another improvement that can be done for the Alpha-Beta agent is to use a smarter Action-Pruning that will reduce the actions pool - and so as the tree size - and by that make our agent faster, and smarter. That could even help us even more by increasing the depth we work with.

With all that said, we should notice that our AQL doesn't really win conclusive percent of the games played (sometimes 60%, sometimes 70-80% but never 90-100%), and that's a testimony of our Alpha-Beta strategy and strength (which we can also see it when the Alpha-Beta wins against the simple agents and some very smart human players very easily and constantly). This is especailly impressive considering the fact that the AQL trained against the AB agent, and so is tailor-designed to try and beat the AB agent.

References

[1] Our project's GitHub: https://github.cs.huji.ac.il/fogrid/AI_project/
[2] https://www.pommerman.com/
[3] https://github.com/BorealisAI/pommerman-baseline
[4] arXiv:1903.08894 [cs.LG]
[5] Bomberman's 2012 AI Project: https://www.cs.huji.ac.il/~ai/projects/2012/Bomberman/
[6] Gao, Chao & Hernandez-Leal, Pablo & Kartal, Bilal & Taylor, Matthew. (2019). Skynet: A Top Deep RL Agent in the Inaugural Pommerman Team Competition.
[7] arXiv:1904.05759 [cs.LG]
[8] arXiv:1811.12557 [cs.MA]