Bubble Bobble AI

20180063 Dohyun Kim

October 4, 2021

0 Github Repository

https://github.com/dohyun1411/Bubble-Bobble-AI

1 Introduction

In 2016, AlphaGo[SHM+16], developed by DeepMind, shocked the world, by defeating Lee Sedol, who was the best professional Go player ever. Not only DeepMind but also other people have been trying to developed AI for various games. AI for Super Mario Bros[BSRC15], one of the Nintendo games, was developed and also for StarCraft[VBC+19], one of the computer games, was developed. In this project, we developed an AI for Bubbule Bobble, one of the most popular platform arcade games, by genetic algorithm(GA). Our final goal is creating an AI that can beat the game faster than human.

2 Game Description

Bubble Bubble is a platform arcade game developed and published by Taito. Player controls a character to move left or right, jump to higher floors, and fall to lower floors. The goal of this game is killing all the enemies on the stage. To kill an enemy there are two steps to do: (1) shoot a bubble to trap the enemy (2) pop that bubble by colliding with them. The key words for this game are as follows.

- Agent: a character controlled by AI, especially GA in this project
- Enemy: enemy to be killed. Enemy can also kill agent by colliding with the agent.
- Bubble: bubble shot by the agent. After shooting, the bubble will go up until it reach to the top
- Trap: trap an enemy by colliding with bubble shot by the agent
- Enemy bubble: bubble that traps an enemy
- Kill: kill the enemy by colliding with enemy bubble

We developed this Bubble Bobble game with PyGame, which is a cross-platform set of Python modules designed for writing video games. We slightly modified and added some constraints to make the problem easier and simpler. The details for the rules are as follows.

- There is only one enemy at the same time.
- Agent should kill the enemy in 10 seconds.
- Once the agent kill the enemy, another enemy will be spawned immediately at random position. Also, the time limit will be set to 10 seconds again.
- There are only two floors on the stage and they are static, i.e., they will not change.
- Bubble can only be popped by colliding with agent who is jumping or falling. It means agent cannot pop the bubble by just moving left or right.

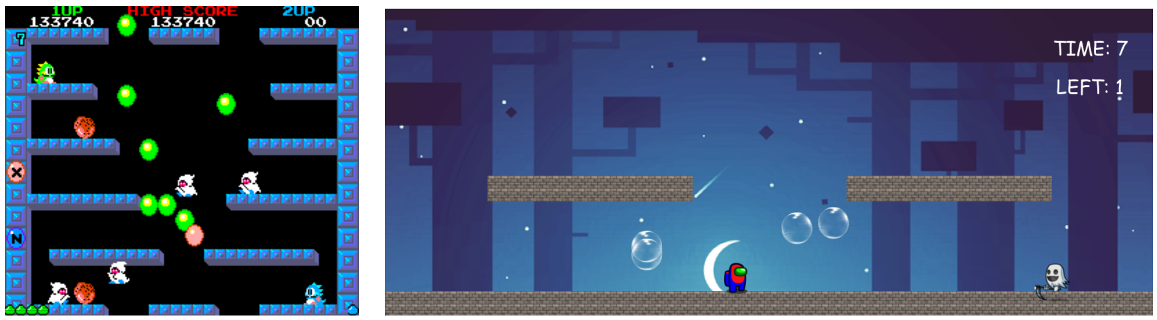

Figure 1: *Left:* Original Bubble Bobble *Right:* Bubble Bobble using PyGame

- Once the bubble reach to the top, the bubble will be popped automatically. If there is an enemy in that bubble, that is, if the enemy bubble reach to the top, the game will be over immediately.
- To simplify the problem, the enemy will shoot automatically when an enemy is detected. Note that, the agent can only shoot the bubble forward.

So, the key points for this game is finding and approaching to the enemy and popping the enemy bubble as fast as possible. Note that, there are only three conditions for game over: (1) agent collides with an enemy (2) time over (3) enemy bubble reaches to the top of the stage.

3 Justification

This section is for justification why we should use metaheuristic algorithm to solve this problem. The key problem in this game is how to approach to the target: either enemy or enemy bubble, fast but safely. So we can consider this problem as one of the path finding problem.

3.1 Traditional Algorithm for Path Finding

Finding shortest path is one of the most famous problem and there are a lot of solutions for it. Dijkstra algorithm is often used to solve the path finding problem. The time complexity of Dijkstra algorithm is known as O(V + E log V), where E and V is the number of edges and vertices, respectively. Even though it is neither in NP-hard nor NP-complete, people have been trying to solve this problem faster than Dijkstra algorithm. This is because to apply Dijkstra algorithm in the real world, we first discretize the world and it will create a huge amounts of edges and vertices, which means it will be time consuming. So many heuristic algorithm was invented as alternative such as A star algorithm or rapidly-exploring random tree(RRT). In this project, we uses GA as alternative.

3.2 GA for Path Finding

In this game, there are some constraints we should consider.

- The target is not static, but dynamic. Moreover, the target act completely randomly.
- Since there is the gravity on the stage, there will be some constraints to move.
- The agent can jump whenever, but cannot fall down if there is a floor below. This makes a directed graph as Figure 2.
- The agent should set proper strategy according to the target, either enemy or enemy bubble.

The last one, setting proper strategy, is important since if the agent just go toward to the target, the agent cannot kill the enemy. To trap the enemy, the agent should approach horizontally, not vertically, because the agent can only shoot bubbles forward. Of course, the agent should be careful of enemy coming from back. To kill the enemy, the agent should pop the enemy bubble. Since bubble can only be popped by jumping or falling of the agent, the agent should approach it vertically.

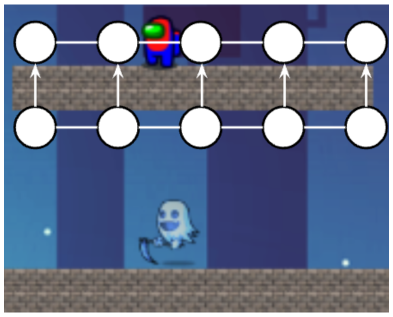

Figure 2: A directed graph will be created since the agent can jump whenever but cannot fall down because of the floor.

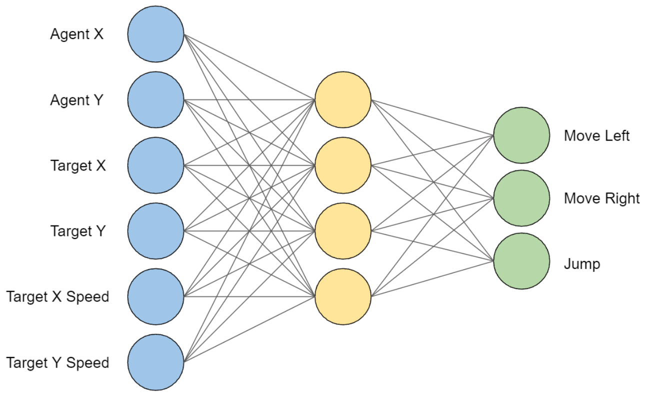

Figure 3: NN model architecture

Since the target and agent are moving over time we should find the path at every unit time. Since the FPS of this game is 60, we should find the path 60 times per one second. Moreover, we should consider all the conditions above. Hence, path finding by traditional algorithm will be slow and cannot suffice. We used GA to find the best path considering all the conditions. The key idea of the GA is setting fitness function related to elapsed time to kill enemy.

4 Artificial Neural Network for GA

We used Neural Network(NN) to control the agent and trained NN by GA. NN needs some information about the agent and the target as inputs to return output: how the agent should act at this time. The parameters of NN is treated as genomes and these genomes will evolve by GA to become better.

Simple fully connected layer, which has only one hidden layer, is used. It has 6 neurons as input layers, 4 as hidden layers, and 3 as output layers as Figure 3. Input and hidden layers have Leaky ReLU as the activation function and the output layer has Softmax function to return the output values.

4.1 Input Layer

6 input neurons are needed for NN. The first two is for position of the agent, x and y, respectively. Third and fourth one is for position of the target, either enemy or enemy bubble. Since enemy and enemy bubble cannot appear at the same time two, not four, neurons will be enough.

However, as mentioned above, setting proper strategy according to the target is important. So the agent should distinguish the target. Since the position of target cannot give the information about what the target is, the agent need more information about target. Since the speed depends on the target, we hope the agent distinguish the target by giving the speed of the target as input neurons.

4.2 Output Layer

There are six possible actions for the agent: (1) stop (2) move left (3) move right (4) jump (5) move left and jump (6) move right and jump. To cover all these actions, one possible solution is prepare six output neurons and treat as classification task. But we wanted to reduce the number of parameters, so we just kept three output neurons for (1) move left (2) move right (3) jump. Actually, we can cover all six possible actions with these three neurons if we do not treat as classification task. Each neuron is treated as a control button. For example, if the first neuron has a value larger than some threshold, we push the move left button, which leads to make the agent move left. If the first and second neurons are activated at the same time, we push both move left and right button, which make the agent stop.

But there are some redundancies because it produces $2^3 = 8$ possible actions while we only need 6 actions. In specific, stop action and jump action are repeated twice, respectively. Stop action can be produced by (1) no neuron is activated (2) first and second neurons are activated, and jump action can by produced by (1) third neuron is activated (2) all the neurons are activated. To solve the redundancies, we set the threshold to 1/3. Since we apply Softmax function at the output layers, the summation of three neurons should be 1. So, setting the threshold 1/3 will guarantee that at least one neuron should be activated. This leads to remove redundancy for stop action. In addition, if we want to activate all three neurons then the value of all neurons should be 1/3, which cannot happen in practice. So, it leads to prevent redundancy for jump action. The mapping from neurons to actions are as follows.

1. No neuron activated - Cannot happen theoretically
2. First neuron activated - Move left
3. Second neuron activated - Move right
4. Third neuron activated - Jump
5. First and second neurons activated - Stop
6. First and third neurons activated - Move left and jump
7. Second and third neurons activated - Move right and jump
8. All neurons activated - Cannot happen in practice

5 Genetic Algorithm

5.1 Hyperparameter for GA

Genetic algorithm is used to train the NN. The hyperparamter for our GA is as follows.

- NPOP: the number of population, this will be fixed for all over generations
- NBEST: the number of genomes to keep in order of fitness
- NCHILD: the number of children produced by the crossover
- NMUT: the number of mutated genomes
- PMUT: the probability of mutation

5.2 Flow of Genetic Algorithm

We applied GA as follows.

1. Initialize population with size NPOP
2. Play the game with genomes and get fitness values.
3. Keep NBEST genomes in order of fitness. We will call it as best genomes.

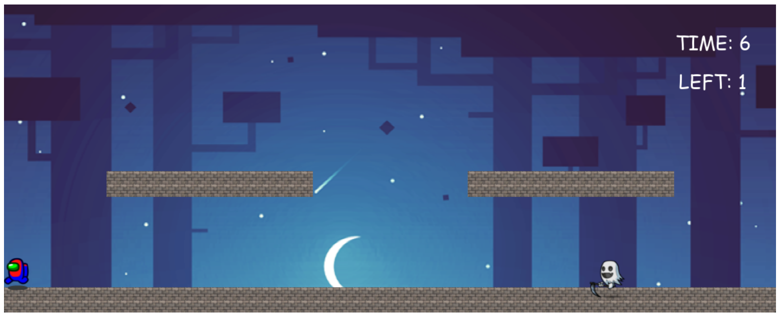

Figure 4: The agent is stuck in the side.

4. Do crossover NCHILD times using best genomes.
5. Do mutation NMUT times using best genomes and children produced by 4 with probability PMUT.
6. New genomes for next generation are composed of NBEST best genomes, NCHILD children, and NMUT mutated genomes.
7. Repeat from 2 until we get a good genome.

By setting NUMT = NPOP - NBEST - NCHILD, we can fix the NPOP for all over generations.

5.3 Fitness Function

Fitness function is calculated as follows.

$$fitness = min\{6000/\sqrt{\Delta t_{trap}}, 750\} + min\{4000/\sqrt{\Delta t_{kill}}, 500\} + \epsilon - 200 \times s \tag{1}$$

Δt_{trap} is the elapsed time from right after killing the enemy to time trapping next enemy. Δt_{kill} is the elapsed time from right after trapping the enemy to time killing that enemy. Since we want to make the agent with small Δt, the fitness function is set to proportional to $1/\sqrt{\Delta t}$. The reason why we should take square root at the denominator is the fact that reaching to the target is more important than fast reaching. So we want to give some fitness values if the agent success to catch the enemy even though it took a long time. Also, you can see the *min* function in the fitness function. This is for upper bound for the values. In addition, the coefficient for Δt_{trap} is larger than Δt_{kill}. This is because once the agent trap the enemy, the enemy bubble immediately appear at that position. So it is easy to approach to the enemy bubble. However, once the agent pop the enemy bubble, a new enemy will be spawned at random position. Finding the path to the random position is difficult. So, we set larger coefficient and upper bound for hard task and lower coefficient and upper bound for easy task.

ϵ guides the agent to approach to the target. It is not a constant but a function affected by the agent position and target position. If the agent moves towards the target, ϵ will be increased, otherwise it will be decreased. This will help agent to approach to the target at the very beginning of the game. Once the agent reached to the target and trap the enemy, ϵ will be meaningless because the fitness value of trapping is much higher than ϵ. ϵ is small enough so that it can only affect at the very beginning.

s is 1 if the agent is at the side when the game is over, otherwise 0. Here, side means either very left or right of the screen. To be simple, check whether the agent is at the side when the game is over and give penalty of -200 if it is. This is because we want to make the agent avoid stuck in the side. Here, stuck means that the agent is doing only one action so the agent cannot escape a certain position. For example, if the agent always move to left as Figure 4, the agent cannot escape the left bottom position.

5.4 Initialization

Initialization of neural network means initialization of parameters of weight matrix. Simply sampling from the Gaussian distribution and making weight matrix with them will be applied for the initialization.

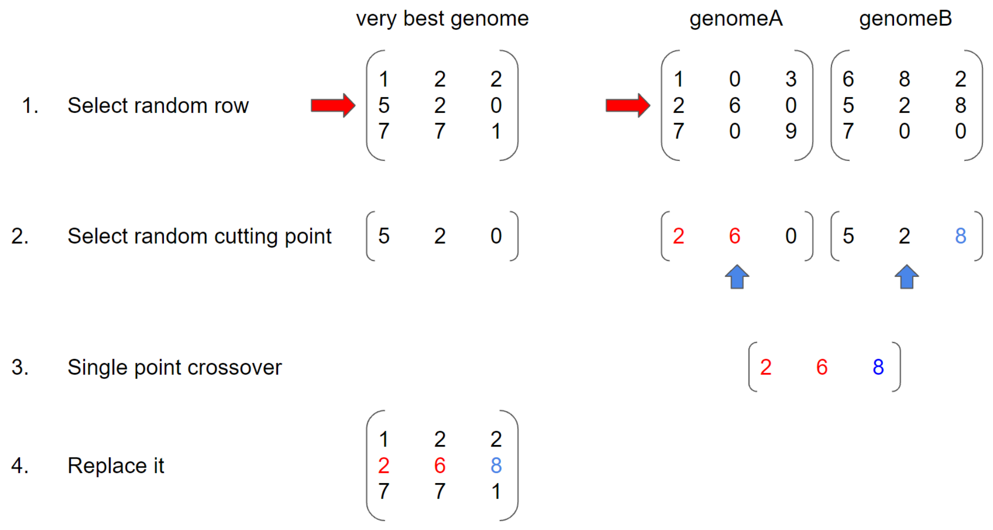

Figure 5: Crossover

5.5 Selection

After creating initial population, the agents will act by the neural network. After finishing the game and getting fitness values, we simply only keeps NBEST best genomes in order of fitness. We will call it as best genomes.

5.6 Crossover

Choose the very best genome, i.e., genome that has the best fitness. The very best genome will produce children by applying cross over with other (NBEST - 1) genomes in best genomes. The child genome produced as follow flows.

1. Prepare the very best genome.
2. Select random two genomes from (NBESET - 1) best genomes. We will call them as genomeA and genomeB, respectively.
3. Select random row from the weight matrix of the very best genome.
4. Select corresponding rows from genomA and genomeB and apply single point crossover with random cutting point.
5. Replace the selected random row of the very best genome to the row produced by 4.
6. Repeat from 3 until all the weight matrices are updated.

For the details please refer Figure 5. This process will be repeated for NCHILD times to produce new NCHILD genomes.

5.7 Mutation

Mutation is applied to best genomes and children genomes. So, there should be (NBEST + NCHILD) genomes to be mutated. Mutation will be applied with probability PMUT and repeat NMUT times to procue NMUT mutated genomes. To keep the POP over all generations, NMUT is set to be (POP - NBEST - NCHILD). So the genomes for next generation are composed of NBEST best genomes, NCHILD child genomes, and NMUT mutated genomes.

Mutation is performed for each row of weight matrix. Each row has a chance of mutation with PMUT. The mutation is implemented by addition of the row, which we want to mutate, weighted by Gaussian distribution. The detail Python code is as follows.

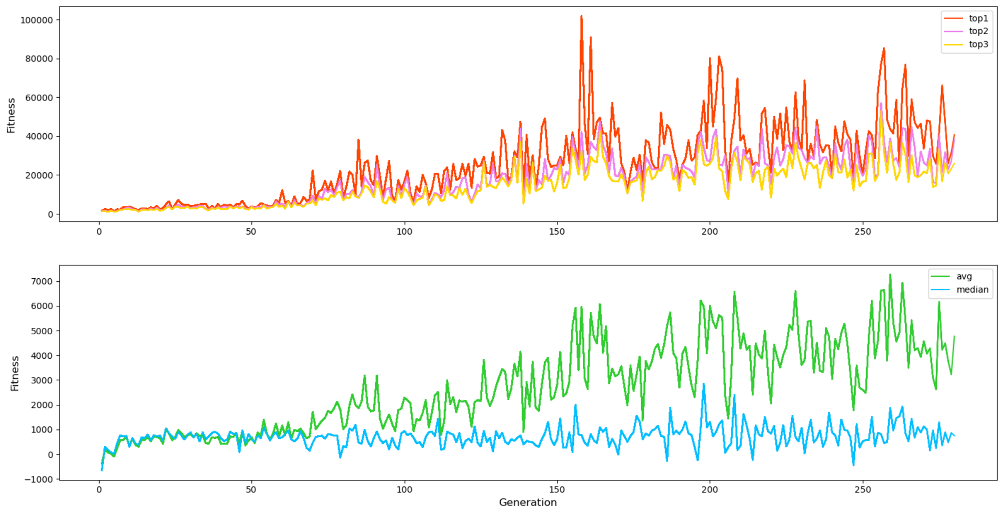

Figure 6: Plot of fitness vs generation

```
    for i in range(genome.w.shape[0]): # for each row of weight matrix
        if random.uniform(0, 1) < PMUT: # with probability of PMUT
            genome.w[i, :] += genome.w[i, :] * np.random.randn(genome.w.shape[1])
```

This mutation progress will be applied for all the weight matrices.

6 Result

The hyperparameter settings are as follows.

- NPOP = 50
- NBEST = 5
- NCHILD = 5
- NMUT = 50 - 5 - 5 = 40
- PMUT = 0.2

The plot for fitness values over generation is as Figure 6. The top subplot indicates the top 3 values in order of fitness. The red line indicates the best genome, the pink line indicates the second best genome, and the yellow line is for the third best genome. The bottom subplot is for mean and median fitness of all NPOP genomes. The green line is for mean, and the blue one is for median. The median value seems like there is no evolution and the mean value seems like slightly improved. However, we are interested in the top genomes so it does not really matter.

As you can see in the top subplot, the top 3 values does not seems to be improved at the beginning. However, from the 70 generation, it started to evolve noticeably reaching over 20,000 fitness value. Since we can approximate the number of killed enemies by $fitness/1000$, it means there are about 20 enemies killed by the agent. As more generations go by, the genome tends to reach higher fitness values and finally got over 10,000 fitness, which means about 100 killed enemies, at the 160 generation. However, after 160 generation, it does not seem to be improved more. It just oscillate even though we trained over 100 generations more.

Here are some strategy of agents over generations we observed. At the generation about 100, the agents seem to know that jumping is somewhat important. So they tried to jump all the time. This might be reasonable because to pop the bubble, the agent should jump or fall. However, jumping all the time will deter to approach to the target. For example, if the target is at the bottom of the stage, the agent should not jump to catch the target. So it is more reasonable if the agents can jump only when they need.

From 100 to 150 generations, some agents walking around within some local area and catch the target when the target is approaching to that local area. This might be reasonable since moving toward to the target, especially enemy, is dangerous. However, this strategy cannot be optimal because there is a time limitation.

At the end of the generation, they have improved version of previous strategy. They are walking around within some local area but moving toward to the target if they think they can reach to the target safely. To move safely, they do some special actions such as moving back when enemy is falling from the upper floor to avoid collision, or keeping some distance between the target and the agent until they think they are safe. Moreover, they are trying to trap and pop the enemy at the bottom of the stage. This is very wise strategy because if the agent trap the enemy at the top floor, the created enemy bubble will have small distance from the top of the stage. Note that, if the enemy bubble reach to the top of the stage, the game is over. So, to make the distance from the top of the stage bigger, trapping and killing the enemy at the bottom of the stage is a good strategy.

7 Discussion

Contrary to our expectations, the fitness values does not increased after some generations, which means more generations does not guarantee that better result. We think that this is because the agent finally get close to the optimal point. However, there is some problem of that agent yet. For example, the agent cannot avoid the enemy when the agent is falling. We know that this is hard task because the gravity constrain the behavior of the agent. We think that to improve the agents, some sensitive mutation will be helpful. For instance, if the fitness value is less than some threshold, mutation will be applied as above mechanism. However, if the fitness value is larger than threshold, it means that we do not need to modify a lot. So, mutation over each row of weight matrix will not be the proper one. Since we just need some small changes, we can apply mutation over each element of weight matrix.

One of the most hard part of this project was even though we set all settings to be identical including hyperparameters and neuralnet architecture etc, it does not guarantee that the same output will be returned. So, to get a good agent, we should repeat the experiment until we find it.

We solved the problem only for simplified version, such as only one enemy at a time. But we also tried to solve more complex problem like multiple enemies at a time, changing of the floor of the stage over time, or increasing of the stage size. However, since the problem was constructed depending on randomness, it was very hard task to find optimal point. But here are some ideas to solve more complex problem for future works.

If the floors of the stage changed, the agent cannot recognize them. So, the agent should try various actions to getting information of the floors. For instance, if the agent jumped but the position does not changed, it means that there is no upper floor above. On the other hands, if the position changed, especially y position, it can conclude that there is an upper floor above. By giving the agent its speed, they could know whether they jumped before or not. So, we need two more input neurons, one is for x speed of the agent and the other is for y speed of the agent.

Handling multiple enemies is tricky but important problem to solve. One simple way to solve this is just extend the input neurons as much as the number of enemies. For example, if there are two enemies at the same time, 8 input neurons are needed for enemies information. Of course there should be more input neurons for the agent. However, since the fully connected layer can only have fixed input layers, if the number of enemies differ from the time, this way cannot be applied. Another way is set the target only one and treat remains as obstacles. For instance, if there are three enemies choose an enemy and set it to be the only target. Then, only 4 input neurons are needed for enemy information as we did in this project. Remaining two enemies are treated as obstacles. The agent needs neither position nor speed of them because the agent just want to get far from them in some way. This means that just brief information would suffice. So, the agent might check which direction and how many the obstacles are in and use this information as input neurons. This will make the nerualnet with fixed size of input layers. We hope that these idea leads to solve more complex problem for future works.

References

[BSRC15] Alejandro Baldominos, Yago Saez, Gustavo Recio, and Javier Calle. Learning levels of mario ai using genetic algorithms. In *Conference of the Spanish Association for Artificial Intelligence*, pages 267–277. Springer, 2015.

[SHM$^+$16] David Silver, Aja Huang, Chris J Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, et al. Mastering the game of go with deep neural networks and tree search. *nature*, 529(7587):484–489, 2016.

[VBC$^+$19] Oriol Vinyals, Igor Babuschkin, Wojciech M Czarnecki, Michaël Mathieu, Andrew Dudzik, Junyoung Chung, David H Choi, Richard Powell, Timo Ewalds, Petko Georgiev, et al. Grandmaster level in starcraft ii using multi-agent reinforcement learning. *Nature*, 575(7782):350–354, 2019.