BTDB2 Eco Simulator Documentation

redlaserbm

April 2023

1 Introduction

Hi there! Welcome to the Bloons TD Battles 2 Eco Simulator! I'm redlaserbm, the code's main developer. The goal of this project is to develop a program that can quickly and accurate simulate eco/farm/alt-eco strategies within b2 so that players can more quickly theory craft and optimize their strategies in-game. Note, however, that this project is all back-end code, with no fancy UI of any sorts associated with it. With this in mind, the following readme is primarily aimed at:

1. Front-end developers who want to develop a UI for the project so that the common man can operate the simulator with ease.
2. Technical audiences with a strong math/coding background who wish to utilize the code for advanced purposes that the common player is unlikely to encounter.

2 Operating the Code

2.1 Installation

Users unfamiliar with coding who just need essential functionality should use the lightweight version of this simulator on spoonoil's website `https://b2.lol`. Newbies unfamiliar with coding who nonetheless want to operate the back-end code should follow the steps below to get the code up and running:

1. Download the latest Anaconda distribution to your computer here. The Anaconda distribution is sufficient to run Python code on your computer
2. Access the repository here. Use GitHub Desktop to clone the repository to your desktop.

3. One of the programs bundled with the Anaconda distribution is jupyter notebook. Launch jupyter notebook, navigate to where you cloned the repository, and open `examples.ipynb`

4. You are now ready to operate the code!

2.2 Operation

In general, the procedure for simulating a game looks like this:

1. Set the round lengths by creating an instance of `Rounds(stall_factor)`. `stall_factor` is a float from 0 to 1 indicating the level of stall in the game (higher value means more stall).

2. Define the buy queue and eco queue for your game state. These queues are lists containing the eco flowchart and flowchart of purchases you intend to follow as you progress through the game.

3. Declare the initial state of the game. How much cash do you have? Eco? Current round? etc.

4. Create an instance `game_state` of `GameState` using the info defined above, and then use `game_state.fastForward(target_round = X)` to simulate what would happen if the game were to progress to round X according to your strategy.

2.3 Features

1. **Simultaneous simulation of eco, farms, and alt-eco**: When given an eco send to use and some arrangement of farms and alt-eco, the simulator accurately tracks the progression of the player's cash and eco over time. The results of the simulator are nearly true to the game.

2. **Easy operation**: Simply input your initial cash and eco, the round to start on, and the purchases you intend to make and the eco flowchart you intend to follow over the course of the match. The code runs in one click and delivers results in seconds.

3. **Complete Farm support**: The simulator supports IMF Loans and Monkeyopolis. Also, the simulator supports compound purchases, such as selling into Monkey Wall Street.

4. **Strategy Comparison**: You can compare multiple different strategies, see how cash and eco progresses over time for each one, and decide what strategy you like better with the `compareStrategies()` function.

5. **Advanced Optimization Potential**: The code's fast run time means that it operates well when used in black-box optimization problems.

3 Advancing the Game State

To simulate a game state along the time interval $(t_0, t]$, where t_0 is the initialized time of the game state,

The method `advanceGameState()` attempts to simulate the game from the associated class's current time until a designated target time. However, this method can only perform accurate simulation provided that nothing about the player's sources of income changes — see Section 3.2 for more information. If something does change, it terminates early. To work around this issue, the method `fastForward()` repeatedly runs `advanceGameState()` over small intervals until a designated target time is reached. In the following subsections, we describe how the simulation works, first by describing the functionality of `advanceGameState()`, and then motivating the `fastForward()` method.

3.1 Computing Payments

`advanceGameState()` begins its simulation by first computing the times and that each income source (eco, farms, boat farms, etc.) pays out and the amount of money that will be granted at each time. The payouts are collected in a list called `payout_times`, and once all payouts projected to occur in the simulation are to occur, the list is sorted by time, with earlier payouts coming first.

In most cases, the exact amount of money to be paid out can be determined during this process, and entries in `payout_times` thus usually consist of a dictionary with three keys, `Payout Type, Payout Time, Payout Amount`. However, this is not possible with bank interest or eco payouts. In all cases, however, `payout_times` is a list of dictionary objects, each with a payout time and instructions to compute the payout amount (if the amount hasn't already been computed).

3.1.1 Computation of Farm Payouts

Different farms pay out at different times, so to compute farm payouts, we iterate on a for loop through each farm, computing the payouts it will give and at what time. For each farm in the for loop, we run a while loop whose general functionality is described by the pseudo-code below to compute its payments over the course of the simulation:

Generally speaking, for each round the simulation covers, we compute the times of all payouts the farm will give in that round. The while loop continues until it detects a farm payout that is to occur after the end of the simulation. It is important to note here that in practice, `advanceGameState()` will be run repeatedly over small intervals, thus it is important to reduce redundant computations as much as possible. The computation of j prevents the code from repeatedly trying to add the same payment over and over again to the list of payouts.

An analogous version of j to prevent repeated attempts of adding payments that are too late

Algorithm 1 Calculate the payout of a Monkey Farm

```
round = self.current_round
while round ≤ target_round do
   j = Smallest j such that the farm's jth payment will appear during the
round (and after self.current_round)
   k = # of payments the farm will give during the round
   for i = j, j + 1, ..., k do
      farm_time = time of farm's ith payout in the round
      if self.current_round < farm_time <= target_time then
         payout_entry = (farm_time, farm_payout)
         payout_times.append(payout_entry)
      else
         BREAK FROM WHILE LOOP
      end if
   end for
   round += 1
end while
```

to the payout list is not necessary because the code automatically breaks the while loop upon detection of such a payout, but what *is* necessary is a system to prevent a farm awarding its kth payout in a round when it cannot do so because of being purchased mid-round. This is the reasoning behind the variable k.

3.1.2 Computation of Boat Farm Payouts

A substantial simplification relative to the algorithm for farms can be made for boat farms because all boat farms award payment at the start of a round. An important consideration that must be made, however, is that Trade Empire buffs the payouts of merchantmen & Favored Trades by $5\% *$ (number of non-T5 boat farms)

Algorithm 2 Calculate the payout from all Boat Farms

```
multiplier = computeTempireBuff()
boat_payout = 0
for boat_farm in boat_farms do
   boat_payout += multiplier*boat_farm.payout
end for
round = self.current_round
while round ≤ target_round do
   payout_entry = (self.rounds.getTimeFromRound(round), boat_payout)
   payout_times.append(payout_entry)
   round += 1
end while
```

3.2 Awarding Payments

Once the list `payout_times` with a sorted list of payments, the code runs a for loop through the payments, awarding them one-by-one until they all have been granted. This is *generally* how the code operates, except that the code cannot accurately finish simulation if the player makes *any* transactions that impact their income. Suppose for instance that in a simulation from the start of Round 12 to the end of Round 14, the player upgrades their 200 Monkey Farm to a 300 farm. Then, in order for the code to accurately simulate the game from the time of this upgrade purchase to the end of Round 14, we must go back and recompute the payouts that this farm will give. Our solution to this problem is to simply terminate `advanceGameState` upon the time of purchase and re-run it again. To implement this solution, we check after every payment in `payout_times` whether the player has enough money to make a purchase — if multiple payments occur at exactly the same time, we process all payments first before runnning this check. If they do, we make the purchase and terminate `advanceGameState` early. We also must account for times when the player wishes to change eco sends. To resolve this issue, we check prior to simulation whether an change in eco sends is to occur before the target time. If yes, then we choose to terminate `advanceGameState` at the time of the eco change and update the eco send the player uses at the end.

To sumarize, the general procedure goes like this:

1. Determine the impact of eco'ing on the player's cash and eco in the time between the previous payment (or the current game state time, if this is the first payment) and the payment we are about to process.
2. (Compute the payout, if necessary.) Award the payment to the player.
3. If the next payout is to occur at a future time — this will almost always be the case — check whether the player can make a purchase in their buy queue. If they can, perform the purchase, and terminate `advanceGameState` early.

3.2.1 Awarding Eco Payments

Eco payments are one of the few sources of income whose payout amount cannot be computed ahead of time during the payout computation process. Assume that the player has an eco send $S = (S_{cost}, S_{eco})$ which on average costs S_{cost} every 6 seconds to use and gives S_{eco} eco every 6 seconds. Next, suppose that there are two payments to occur at times t_1 and t_2 respectively with $t_1 < t_2$. Let C_i, E_i denote the cash and eco at time t_i. Then, since $t_1 + 6 \geq t_2$, we have

$$C_2 = C_1 - \min\left(\frac{C}{S_{cost}}, \frac{t_2 - t_1}{6}\right) S_{cost} \tag{1}$$

$$E_2 = E_1 + \min\left(\frac{C}{S_{cost}}, \frac{t_2 - t_1}{6}\right) S_{eco} \tag{2}$$

3.2.2 Loan Payments

The accurate awarding of payments is no longer straightforward if the player has outstanding loans as a consequence of activating IMF Loan. Assume that the player has C_1 cash and L_1 dollars of outstanding debt from IMF Loans and is about to receive a payment of P dollars. Then, the updated cash and loan amounts C_2 and L_2 the player will have after this payment is received is given by

$$C_2 = \begin{cases} C_1 + P/2 & \text{if } P/2 \leq L_1 \\ C_1 + P - L_1 & \text{if } P/2 > L_1 \end{cases} \tag{3}$$

$$L_2 = \begin{cases} L_1 - P/2 & \text{if } P/2 \leq L_1 \\ 0 & \text{if } P/2 > L_1 \end{cases} \tag{4}$$

3.3 The Buy Queue

(I will expand on this subsection more later.) The buy queue `self.buy_queue` in the `GameState` class is a list whose entries describe purchases the player wishes to make over the course of the simulation. To account for the possibility of compound purchases, such as selling into Monkey Wall Street, each element in the buy queue is itself a list in fact. After all purchases in a given time are made, the code checks whether the first item in the buy queue can be performed. If it can be done, it will be done, and `advanceGameState()` will terminate early in this case.

In general, in the presence of IMF Loans, the order in which transactions are performed *does* matter.

3.4 Simulation in Practice

The early termination feature of `advanceGameState` is necessary for accuracy but is inconvenient for the user because regardless of how far one instance running of `advanceGameState` progress, they want the code, when it runs, to advance to precisely the target time that they specify! This motivates the existence of the `fastForward` method, which repeatedly runs `advanceGameState` until the target time is reached.

Something we must be careful about when repeatedly running `advanceGameState` is the possibility of wasted computation time. For example, if we simulate a game state from the start of Round 12 to the end of Round 14 and purchase occurs say a few seconds into Round 12, then *all* of the computation time spent on payments occurring after that purchase will be wasted. To prevent a substantial amount of computations from being discarded, we run the code in small intervals. Note that as a consequence that we must take care to ensure that `advanceGameState` does not perform redundant computations, or else running the method many times like this repeatedly will slow down runtime — see Section 3.1.1 for an example of such care being taken.

An added bonus of running the code repeatedly is that, since cash and eco are logged every time the simulation terminates, the resulting log of the player's cash and eco over time will be more detailed than just one pass of the method.

4 Feature Requests / Features to add

4.1 Alt Ecos

Still need to add support for druid and heli alt-ecos. (Expected date of implementation: May 10th).

4.2 Simplifying Code Structure

Currently, the entirety of the code is contained in `main.ipynb`. It'd be nice if the examples section of this notebook was moved to a different file.

4.3 Optimizing the buy queue

Currently, the code, when checking whether a purchase in the buy queue can be made, re-computes the cost of this operation from scratch each time it checks. However, if the impact of the purchase to the player's cash and loans does not vary over time, we can save computation time by computing the cost once, storing it, and then reaccessing it on subsequent checks.

An exhaustive list of time-dependent payouts include the following:

1. Sell operations when the player has debt: The amount of cash the player will actually get from the payment depends on his loan value as evidenced by equation 3. Whenever the player receives a direct payment, we must recompute the impact the sell operation would have on his cash & eco because of the possibility of switching from one case to the other in the computation of resulting cash.
2. Bank withdrawals: As time passes the bank will accumulate more and more money

4.4 Robust Logging

The `GameState` class is equipped with a robust logging feature which, after the player runs a simulation, allows the player to see what payments were given and from what sources.

However, we also need a nice logging feature that works for when comparing *multiple* strategies against each other.

4.5 More Detailed Stats

When comparing strategies, it'd be nice if users could see more than just cash and eco over time.