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. This document contains a high-level explanation for how the code works, and is written with the following audience in mind:

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. Back-end developers who wish to rapidly get up to speed on how the code functions and ultimately contribute lines of code.
3. Practitioners with a strong math/coding background who wish to utilize the code for advanced purposes that the common player is unlikely to encounter.

2 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 2.3 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.

2.1 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).

2.1.1 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:

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
```

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 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.

2.1.2 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
```

2.2 Computation of Supply Drop Payments

For simplicity sake, we assume that supply drop/elite sniper actives are *always* used as soon as they possibly can. This is optimal except when the player does not yet have an elite sniper. (A solution to this shortcoming

2.3 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.

2.3.1 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

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

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

2.3.2 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

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

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

The global function `impact(cash, loan, payment)` is used to compute the resulting cash and loan amounts had from receiving a payment possibly in the presence of outstanding debt.

2.4 The Buy Queue

The *buy queue* represents the list of purchases the player intends to make over the course of the simulation. Generally, this represents the farming flowchart the player will adopt, but it also can encompass buying/selling defense and alt-eco. The buy queue is stored in the `GameState` class as `self.buy_queue`, which is structured as follows:

```
self.buy_queue: List: List: dict: {
                                   'Type': str,
                                    ...
                                   }
```

That is, each element in the buy queue is a list of lists, containing `dict` objects that carry instructions on what transactions to perform and in what order to perform them. The complete list of entries in the `dict` object may vary, but all contain a key called `'Type'` which gives instructions for the code on how to read the queue. End users need not be concerned with the structure of the `dict` object but rather the functions they can use which create these dict objects. For a complete list of such functions, go to the section "Functions For Simulation" in `main.ipynb`.

2.4.1 Compound Transactions in the Buy Queue

In most cases, the elements of the buy queue will just be lists of length one, so only one action is performed, but being able to specify multiple actions at once is useful because sometimes players may want to sell into more powerful upgrades and wait until they have the money to do so. An example of this is as follows: Suppose the player has $2 \times 320 + 1 \times 203 + 1 \times 204$ farms and wants to sell into a 205 farm. This transaction would consist of 4 actions:

1. Sell a 320 farm
2. Sell the other 320 farm

3. Sell the 203 farm

4. Upgrade the only remaining farm to 205

If listed separately, the simulator would perform each of the first three actions as soon as they become listed first into the buy queue. However, this is clearly not desirable, as the player would want to hold on the farms as long as possible to get their payments first *before* selling into the Monkey Wall Street.

It's worth remarking here that the order that actions are listed within elements of the buy queue matters! If for example, in the above scenario, we tried to buy the Monkey Wall Street first and then sell the other farms, the code as expected prevents a proper sell-in into the MWS, selling the other farms only after MWS is finally afforded.

From here on out, we'll refer to items in the buy queue as **transactions**. If a transaction has multiple actions listed, we'll refer to it as a **compound transaction**.

2.4.2 Multiple Transactions in the Buy Queue

If the player receives a large lump sum payment, perhaps from a Monkey Wall Street or a bank withdrawal, it is possible that the player could perform multiple transactions within the buy queue. Our solution to this is the following pseudo-code:

Algorithm 3 Process transactions in the buy queue as soon as the player can afford them

```
flag = False
while len(self.buy_queue) > 0 do
   flag = True
   h_cash, h_loan = hypothetical cash and loan amounts after transaction 1
   if h_cash ≥ self.buffer then:
      self.cash = h_cash
      self.loan = h_loan
      Update game parameters
      self.buy_queue.pop(0)
   else
      BREAK WHILE LOOP
   end if
end while
```

The `flag` is used to signal to the code whether a transaction took place. If it did, then after breaking the while loop we terminate `advanceGameState` early in line with the discussion of section 2.

2.5 The Eco Queue

The eco queue represents the strategy of eco sends the player intends to use over the course of the simulation. The eco queue functions similarly to the buy queue, except that unlike in

the buy queue, whether or not a player can switch eco sends is dictated not by money, but by time.

2.6 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 2.1.1 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.

3 Farms

Farms are stored in the `GameState` class as `self.farms`, which takes the following data structure

```
self.farms: dict:  {
            0:  MonkeyFarm object,
             ...
            n-1:  MonkeyFarm object,
            }
```

Notice here that the keys are nonnegative integers, which suggests at first that `self.farms` ought to just be a list of MonkeyFarm objects, seeing that the dict structure used doesn't appear to provide any advantages over a list. In practice, the dictionary structure functions a *lot* smoother than a list structure. To see why, consider an example game scenario where we have the arrangement of farms given by table 1

Naturally, to sell into Monkey Wall Street, we would want to sell farms $0, 1, 2$, and then upgrade farm 3 to a $(2, 0, 5)$ farm. However, as soon as one farm is deleted, the indices of

Index	Type
0	$(3, 2, 0)$
1	$(3, 2, 0)$
2	$(2, 0, 3)$
3	$(2, 0, 4)$

Table 1: Caption

other farms may be shifted as well. Under a list structure, this would *actually* look like selling the farm at index 0 three times and then finally upgrading the farm at index 0 to $(2, 0, 5)$. Although counter-intuitive, this alone doesn't necessarily disqualify the list approach from consideration. However, we also need to check ahead of time that we can actually sell into MWS. For this example, under the list structure, this involves checking the sell values of farms $0, 1, 2$ and the upgrade cost of farm 3. Assuming well-written code to accommodate the list structure, in order for an end user to operate the code correctly in this case, they would have to determine ahead of time that the farm indices to check sell values on is $0, 1, 2$ and the index to check upgrade cost on 3 (easy enough) *and* know that every action has to actually occur on index 0 all the way through (far from easy).

On the other hand, this issue does not occur with the dictionary structure, as the deletion of elements does not cause indices to shift. Thus, the indices we check sell values on correspond to the indices used for sell operations and the index we check upgrade cost on corresponds to the index used for the buy operation.

3.1 Labeling Farms (and other Alt-Eco's!)

The trade-off of using a dictionary with integer keys is that we must determine the key each farm in the simulation is associated with. The code is written so that:

1. The code uses `self.key` to assign keys to farms. This is initialized as the maximum key value among all initialized farms (or 0 if there are none).
2. Whenever a farm is purchased, it is assigned the key `self.key`, then `self.key` is incremented by 1.

When a best-practice of initializing n with the keys $0, 1, \ldots, n-1$ is adopted, this means that index i represents the $i+1$th farm to be purchased in the game.

This same labelling structure is also used for alt-ecos. Again, the idea is that we want to avoid problems associated with the index shifting that occurs when deleting items from a list.

3.2 Banks

In most cases, farms simply function by paying out some fixed amount of money some fixed amount of times per round. Banks function similarly, except:

1. Payments go into the bank's account rather than the player's hands
2. At the start of each round, each bank deposits 400 dollars and awards 20% interest right before awarding its regularly scheduled first payment of the round.

To resolve the matter, we introduce the `MonkeyFarm` class variables `self.account_value` and `self.max_account_value` and the buy queue action `withdrawBank(ind)`. To accurately simulate the above two bank-exclusive behaviors, we do the following:

1. In `GameState.advanceGameState()`, whenever we detect that the farm is a bank, we add the farm's payments to its bank account rather than `GameState.cash`. The player can take out this money by placing `withdrawBank(ind)` in the buy queue.
2. In `GameState.advanceGameState()`, upon the start of a new round, the code appends a payout entry of type `"Bank Interest"`. This payout is written in the code so that, before (and after) sorting, it occurs before the bank's first regular payment of the round.

3.3 IMF Loans

The $x4x$ bank is just like $x3x$ except that it comes with a feature to take out a loan. To track when the loan can be used we introduce the class variable `self.min_use_time`. Generally, a farm will be initialized with this set to `None`, but if upgraded to $x4x$, then we set it equal to `purchase_time + 20`. To use the loan we introduce the buy queue action `activateIMF(ind)`. Because of how loans work, instead of trying to track the amount of money owed to each bank, we can just act like for simplicity that all loans come from the same bank and use the variable `GameState.loan` to track the player's outstanding debt. We refer the reader back to section 2.3.2 for an explanation on how the code handles payments when the player has debt.

3.4 Monkeyopolis

When the player upgrades an IMF loan to a Monkeyopolis it is now optimal to use the ability as soon as it becomes available, and we assume in the code that the player does just that. In $x4x$ farms, `self.min_use_time` is used to check whether the player can IMF Loan when that action arises in the buy queue. However, in an $x5x$ farm, this is repurposed to compute recurring payouts from said farm during the payout computation and awarding process of `GameState.advanceGameState()`.

A key takeaway here is that so-called *optional payouts*, like the IMF Loan, are handled in the buy queue, while *automatic payouts*, like the Monkeyopolis payouts, are handled in the payout computation and awarding process.