A tutorial on sc2reader: events and units

Sep 4, 2018

I want to construct a database for macro learning in StarCraft 2. One way to study replays and mine information from them is using `pysc2`, an API developed by DeepMind that is specially designed for Reinforced Learning. Nahren and Justesen developed a tool for mining replays using this API, but unfortunately the DeepMind's API can only play replays that match the binary version of the game, and StarCraft 2 is being patched regularly, thus many "old" replays render unparsable. The other way to do it is using `sc2reader`, an API developed by 26 contributors including David Joerg, Graylin Kim and Kevin Leung.

In this blogpost I'll start talking about `sc2reader` and the information that can be mined using it. We will focus for now on the events regarding units and postpone the discussion about *spatial* and *command* related events for later posts. Feel free to follow all the code that is presented in this blogpost in this gist notebook (I commented out the print statements).

The replay object

After installing `sc2reader` using `pip install sc2reader`, we can import the API and a replay like this:

```
import sc2reader

replay = sc2reader.load_replay("path/to/replay.SC2Replay")
```

If you're using Windows, you can find your own replays buried deep at `Documents\StarCraft II\Accounts\...`, if you want replays from professional players you can look at Spawning Tool, a website which uses `sc2reader`. Copy some of them in your working directory.

This replay object stores all the information about the file, including for example the participants and *rules* of the game at `replay.attributes`, the players at `replay.players` and the winner at `replay.winner`. Remember that all these methods and attributes can be listed using `dir(replay)`.

We will focus on `replay.events`.

Events in a replay

`replay.events` is a list storing all the events of the game. Among these are UnitBornEvents, UnitInitEvents, UnitDoneEvents, UnitDeadEvents, CameraEvents and many, many more. You could for example figure out all event types in your replay running

```
event_names = set([event.name for event in replay.events])
```

that is, all events have an attribute called `name`, which stores the type of the event. Let's separate events with regard to their types (or names):

```
events_of_type = {name: [] for name in event_names}
for event in replay.events:
    events_of_type[event.name].append(event)
```

`events_of_type` is a python dictionary in which the keys are event names in strings (e.g. `"UnitBornEvent"`), and the keys are lists of all the events of said type, so `events_of_type["CameraEvent"]` returns the list of all CameraEvents.

Let's go more in depth in some of these events:

UnitBornEvents

Say one of the players is a Terran and created a Marine, then a `UnitBornEvent` was issued, storing the location, time, unit type and unit controller, among other things. UnitBornEvents are generated, according to the documentation, *every time a unit is created in a finished state*. This doesn't include then warpgate units and buildings (except perhaps the initial ones).

The most important attributes for our purposes are:

- `frame` and `second`, which store the frame in which the unit was created.
- `unit`, `unit_controller` which store the actual unit (e.g. Zergling) and the player which created it respectively. You can quickly access which player created that unit with `control_pid` which is either a 1 or a 2 (i.e. either player 1 or 2).
- `x` and `y`, which store the location in which this event took place.

For example, try printing all these events using a for loop:

```
unit_born_events = events_of_type["UnitBornEvent"]

for ube in unit_born_events:
    print("{} created {} at second {}".format(ube.unit_controller,
                                              ube.unit,
                                              ube.second))
```

If you do so, you'll notice that the first events have `None` as the unit controller. These are precisely the events that track the creation of the map: mineral patches and Vespene gas geysers.

UnitInitEvents and UnitDoneEvents

If you start building a Supply Depot, a `UnitInitEvent` is issued holding about the same information of UnitBornEvents. It tracks time, location and type of unit created. This event is helpful when tracking buildings.

Try printing these events with

```
unit_init_events = events_of_type["UnitInitEvent"]

for uie in unit_init_events:
    print("{} started creating {} at second {}".format(uie.unit_controller,
                                                       uie.unit,
                                                       uie.second))
```

Once these units finish, a `UnitDoneEvent` is created. One must be careful, because this `UnitDoneEvent` doesn't track the unit controller. Thankfully, every unit has a unique ID, so one could maintain a list of the units that were being created and then just test for pertenence. Let's print the units as they finish:

```
unit_done_events = events_of_type["UnitDoneEvent"]

for ude in unit_done_events:
    print("{} finished".format(ude.unit))
```

UnitDiedEvents

Once a unit dies, a `UnitDiedEvent` is generated. It stores the unit that died, the killer player and killing unit, and also location and time.

```
unit_died_events = events_of_type["UnitDiedEvent"]

for udiede in unit_died_events:
    print("{} was killed by {} using {} at ({}, {})".format(udiede.unit,
                                                            udiede.killer,
                                                            udiede.killing_uni
                                                            udiede.x,
                                                            udiede.y))
```

A bit more about units

In all these events, we have considered objects of type `Unit` (which are stored in `event.unit` for each of these events). Let's talk a little bit more about this object and its atributtes:

- `name`, which is literally a string with the name of the unit.
- the flags `is_army`, `is_building` and `is_worker`, which hold a boolean stating whether the unit is of that type.
- `minerals` and `vespene`, holding the cost of producing that unit.
- killing-related attributes such as `killed_by`, `killing_unit`, `killing_player`, `killed_units`.
- `location`.
- `owner` and `race`.

Writing a worker counter

Let's handle these events and create a worker counter. To do so, note that we only need to consider UnitBornEvents and UnitDiedEvents, because workers always enter the game in a finished state. We will write a function that takes a replay, a second and a player id and returns the amount of workers said player had up to that second. To do so, the function will maintain a list holding the workers that have been created and that haven't died yet.

```
def worker_counter(replay, second, player_id):
    workers = []
    for event in replay.events:
        if event.name == "UnitBornEvent" and event.control_pid == player_id:
            if event.unit.is_worker:
                workers.append(event.unit)

        if event.name == "UnitDiedEvent":
            if event.unit in workers:
                workers.remove(event.unit)

        if event.second > second:
            break

    return len(workers)
```

And, why not, let's use this function to plot a graph of seconds vs. amount of workers:

```
length_of_game = replay.frames // 24
workers_1 = [worker_counter(replay, k, 1) for k in range(length_of_game)]
workers_2 = [worker_counter(replay, k, 2) for k in range(length_of_game)]
```

```
plt.figure()
plt.plot(workers_1, label=replay.players[0])
plt.plot(workers_2, label=replay.players[1])
plt.legend(loc=2)
plt.show()
```

We first get the total amount of seconds by dividing the total amount of frames (i.e. `replay.frames`) by the framerate (24), then we create a list with workers at each second and then we plot it using matplotlib.

Conclusions

We discussed some of the events that are tracked by `sc2reader` in a replay. Namely, we considered the attributes of events that track unit creation and death: `UnitBornEvent`, `UnitInitEvent`, `UnitDoneEvent` and `UnitDiedEvent`. Moreover, we presented the `Unit` object and the information it holds. The possibilities are endless. With this information one could easily track important data such as army spending, worker count, army count, build order and much more.

MGD's blog

MGD's blog
miguelgondu@gmail.com

miguelgondu

The blog of Miguel González, a mathematician with interest in videogames and statistical learning.