P3: Behavior Trees for Planet Wars

Introduction

In this programming assignment you will need to implement, in Python, a bot that plays Planet Wars using Behavior Trees. Planet Wars is a real-time strategy game in which you have to conquer a galaxy, planet by planet. Each planet produces ships per turn, and ships can be used to take over other planets from the enemy or neutral forces (similar to Galcon). The bots will be implemented using a version of this game implemented in Java, which is described in detail in the next sections. Your goal is to design a reactive bot with a single behavior tree which successfully wins against a series of test bots, each representing a unique challenge. Furthermore, we will run a small competition with the class's bots. We will pairwise test the submitted bots using three randomly selected maps per each pair. The winner(s) of the competition will earn extra credit.

Example

In order to run the game and test your bot interactively you need to execute the following command (assuming you are in the /src folder):

```
$ python run.py
```

When you run this code you should see a new window with the initial state of the first match, as shown in Figure 1. This program runs a match between your bot and the five other pre-defined bots that you have to beat. It then replays each match in an interface (so you can watch!). To watch the game you can use the buttons on the window. The match can also be run without the graphical interface (using the *test* function), where it runs a match and reports the result in the console.

If you are having trouble with run.py, look at its lines 11, 12, 23, and 24 and possibly change python to python3 or what-have-you.

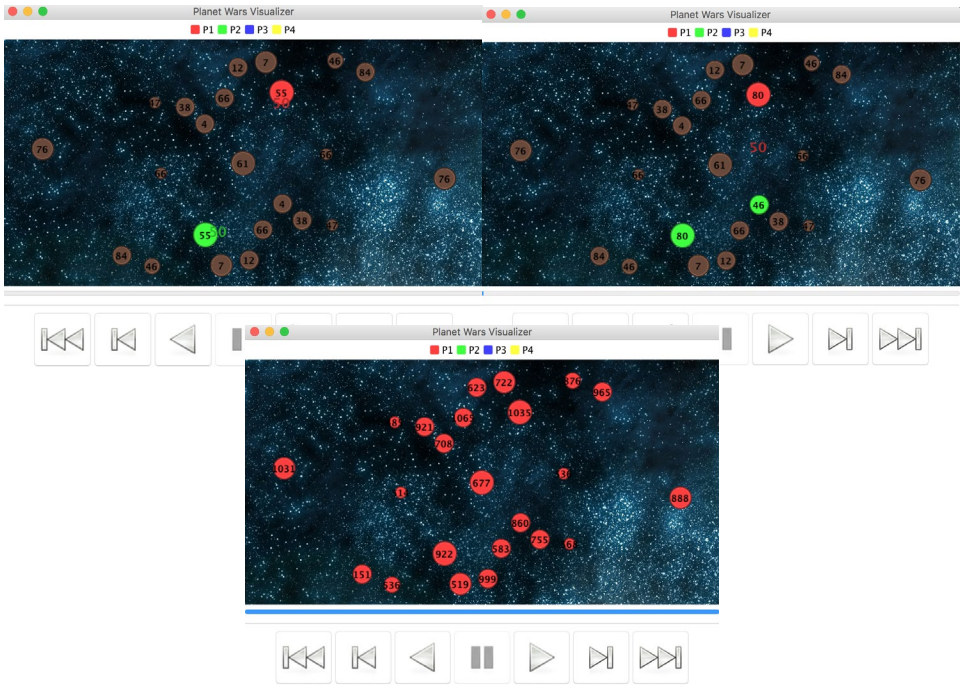

Figure 1: Screenshots of the first match (*bt_bot* vs. *easy_bot*) in three different states: initial state, after 5 ticks and final state (player one wins).

Base Code Overview

- **run.py**
  This is the primary point of execution for your code. *show_match* function runs a match between two bots then replays the match in an interface. *test* runs a match and reports the result in the terminal.

- **behaviour_tree_bot/bt_bot.py**
  This is where you implement your overall strategy to play the game. There is already a basic strategy in place here. You can use it as a starting point, or you can throw it out entirely and replace it with your own.

- **behaviour_tree_bot/behaviors.py**
  Here is where you will implement your functions for action nodes, typically issuing orders. Each function should only take the game state as a parameter. There are two actions already implemented here as examples: *attack_weakest_enemy_planet* and *spread_to_weakest_neutral_planet.*

- **behaviour_tree_bot/checks.py**
  Here is where you will implement your functions for state-based conditional checks. As with actions, each function should only take the game state as a parameter. There are two

conditional checks already implemented here as examples: *if_neutral_planet_available* and *have_largest_fleet.*

- behaviour_tree_bot/bt_nodes.py

Contains all of the node classes needed to build your behavior tree. You will use the Check, Action, Selector, and Sequence classes. **Do not use the Node or Composite classes. These are abstract parent classes used for the other node types.**

- *Check* - a leaf node which contains a check function, i.e. a function which checks for a condition within the state. **These function calls should not issue orders.**
- *Action* - a leaf node which contains an action function, typically issuing one or more orders.
- *Selector* - a branching node containing an ordered list of child nodes. When the selector node is executed, it will attempt executing its child nodes *in order* until one returns a **success** (True). Once a child returns a success, it skips the execution of the remaining child nodes.
- *Sequence* - a branching node containing an ordered list of child nodes. When the sequence node is executed, it will attempt executing its child nodes *in order* until one returns a **failure** (False). Once a child returns a failure, the sequence is aborted.
- ***Optional***: Using the established classes, implement behavior tree nodes for Decorator types (such as Inverter/LoopUntilFailed/AlwaysSucceed/etc) and/or Random Selector.

- planet_wars.py

Contains classes for planets, fleets, and the game state (PlanetWars), as well as two functions (issue_order and finish_turn).

- *PlanetWars* contains all of the relevant information for the game as well as several convenient interface methods for accessing information:
  - my_planets() - list of all planets owned by your bot
  - neutral_planets() - list of all planets not owned
  - enemy_planets() - list of all planets owned by the enemy
  - not_my_planets() - list of all planets not owned by your bot
  - my_fleets() - list of your bot's fleets
  - enemy_fleets() - list of enemy fleets
  - distance(source_planet, destination_planet) - distance between two planets
  - is_alive(player_id) - returns True if player owns any fleets or planets and False otherwise
- *Planet* contains the information describing a planet:
  - ID - the planet's unique ID number (also the index in PlanetWars.planets)
  - x, y - the coordinates of the planet
  - owner - the ID of the owner (0 - neutral, 1 - you, 2 - opponent)
  - num_ships - number of ships

    - growth_rate - how many ships are added each turn if not neutral
- *Fleet* contains information describing a fleet:
    - owner - the ID of the owner (0 - neutral, 1 - you, 2 - opponent)
    - num_ships - number of ships
    - source_planet - where the fleet originated
    - destination_planet - where the fleet is headed
    - total_trip_length - the distance between the two planets
    - turns_remaining - how many turns remain until the fleet arrives
- *issue_order(state, source_planet, destination_planet, fleet_num_ships)*
    - If the source planet possesses enough ships, a new fleet is created headed to the destination planet. The source planet's number of ships is updated.
    - If the source planet is not owned by the bot or if there are insufficient ships for the order, the function returns False (a failure)
- *finish_turn()*
    - Passes the turn to the opposing player

Behavior Trees

Figure 2 shows the behaviour tree for the strategy already implemented in *bt_bot.py*. The code to construct this tree is defined in the *setup_behavior_tree* function.

```
Selector: High Level Ordering of Strategies
| Sequence: Offensive Strategy
| | Check: have_largest_fleet
| | Action: attack_weakest_enemy_planet
| Sequence: Spread Strategy
| | Check: if_neutral_planet_available
| | Action:
spread_to_weakest_neutral_planet
| Action: attack_weakest_enemy_planet
```

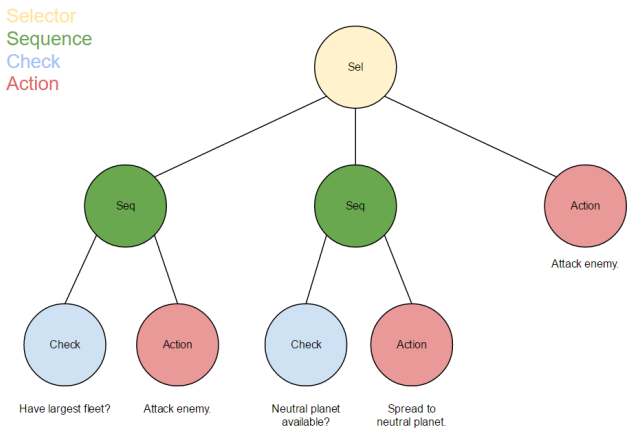

Figure 2: Example of behaviour tree described in text and in graph format.

This tree is executed in the following order:

- *Selector* -> begin executing child nodes
    - *Sequence* -> begin executing child nodes
        - *Check (Largest fleet?)* -> False. Return failure (False).

    - *Sequence* -> Failure received. Abort execution; return failure.
- *Selector* -> Failure received. Attempt next child.
    - *Sequence* -> begin executing child nodes.
        - *Check (Neutral planet available?)* -> True. Return success (True).
    - *Sequence* -> Success received. Execute next child.
        - *Action (Spread to neutral planet)* -> Order issued. Return success.
    - *Sequence* -> Success received. Child nodes have completed. Return success.
- *Selector* -> Success received. Abort execution of remaining children.

Some notes:

- Leaf nodes (Actions and Checks) contain functions for execution
- Composite nodes (Selectors and Sequences) contain child nodes.
- Selectors and Sequences can take an optional parameter, called name, for clarity when inspecting your tree with print(root.tree_to_string())
- A composite node may contain any number of child nodes of any variety.
- **Useful:** Each node type contains a copy() method which returns a copy of the node. If the node is composite, it recursively copies the entire subtree for reuse.

Requirements

- Submit a behavior tree-based bot (*bt_bot.py* in the behavior_tree_bot directory) which successfully wins agains all of the five provided bots. We will run it to verify that it wins.
- Submit the behaviors.py and checks.py files containing the primitive actions and checks used by your bot.
- Ensure that your bot operates within the time requirements of the game (1 second per turn).
- Submit a text file showing your behavior tree (use print(root.tree_to_string()).

Grading Criteria

- Your bot operates within the time constraint of 1 second per turn.
- Your bot completes each test case (individual points per test case).
- The winner(s) of the competition will earn extra credit.

Submission instructions

Submit a zip file named in the form of “Lastname1-Lastname2-P3.zip” containing:

- The zip file should contain your behavior_tree_bot folder, including all of its files. **Please include the folder itself and not simply the files. This will facilitate the automation of the class-wide competition.**
- You should also submit a text file containing the output of tree_to_string() called from the root of your behavior tree. If you have not changed the particular line of code in the original bt_bot, it should be the first item logged after construction of your behavior tree. You can copy it from there to a new text file.

Tips

We've provided an opponent bot, called *do_nothing_bot*, which takes no actions. If you want to observe your bot without the complication of an active opponent, try it out.

Debugging your bot cannot be done with print statements. The game (a Java program) communicates with the bots via stdout. Fortunately, Python has a handy logging module in its standard library for mature debugging practices. Here's an overview:

- At the start of each bot process, the logger creates a log file with the same name via

```
logging.basicConfig(filename=__file__[:-3] + '.log', filemode='w', level=logging.DEBUG)
```

- If your bot crashes, the trace will be recorded by the logger via the logging.exception() call in the try-> except portion of the bot code.
- While the process is running, any logging call in *any module* will be written to this file. These are marked by what type of logging call was made (info, warning, debug, error, exception, etc --- see https://docs.python.org/3/library/logging.html) For example, we have the bt_bot set up such that when the behavior tree is constructed, it is logged with

```
logging.info('\n' + root.tree_to_string())
```

- We have already wrapped the execution calls of the behavior tree in logging.debug() calls, and you should see the execution trace in the logs by default.
- You may additionally want to add your own debug calls in your behavior and checks files. If you're getting incorrect logic or values, try logging.error() upon receiving invalid behavior.

References

- An overview of behavior trees: http://gamasutra.com/blogs/ChrisSimpson/20140717/221339/Behavior_trees_for_AI_How_they_work.php
- Decorators - http://aigamedev.com/open/article/decorator/
- Description of the game: http://franz.com/services/conferences_seminars/webinar_1-20-11.gm.pdf