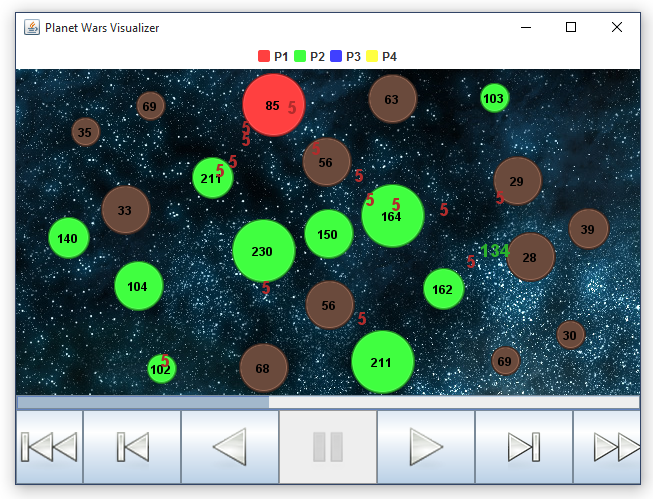
**P4: Behavior Trees for Planet Wars**



**Objectives**

* Learn the basics of constructing behavior trees
* Understand how various task environments influence the design of a reactive bot

**Assignment Requirements**

* Submit a behavior tree-based bot (*bt\_bot.py* in the behavior\_tree\_bot directory) which successfully completes all of the included test environments.
* Provide encapsulated behavior and logic checks for the design of said bot.
* Ensure that your bot operates within the time requirements of the game (1 second per turn).

**Helpful Resources**

* 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://planetwars.aichallenge.org/problem_description.php>

**Base Code**

*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

*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.

Example:

Selector: High Level Ordering of Strategies

| Sequence: Offensive Strategy

| | Check: have\_largest\_fleet

| | Action: attack\_weakest\_enemy\_planet

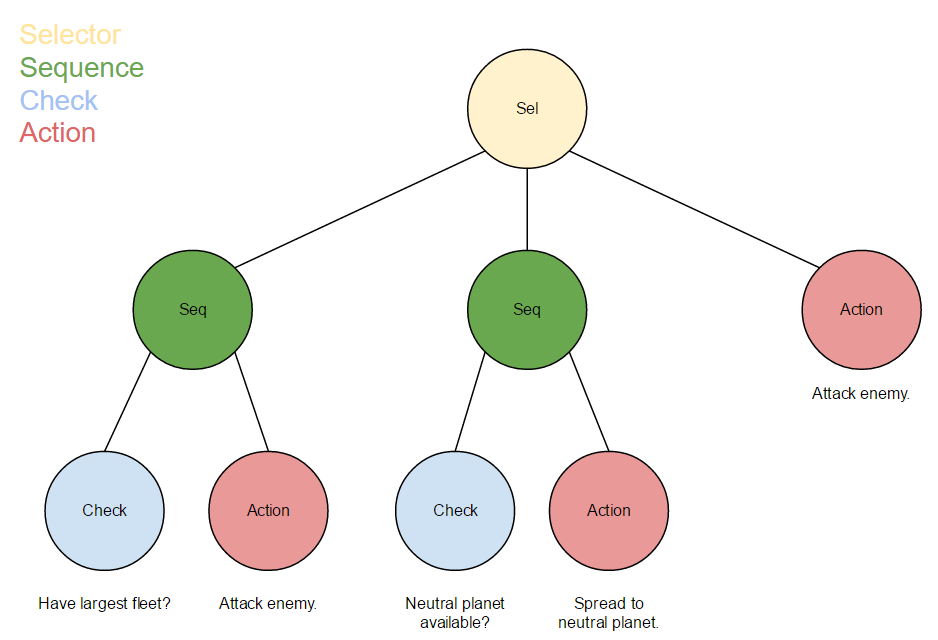
| Sequence: Spread Strategy

| | Check: if\_neutral\_planet\_available

| | Action: spread\_to\_weakest\_neutral\_planet

| Action: attack\_weakest\_enemy\_planet

Visual representation:



Example execution trace:

* 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.

*behaviors.py* - Contains your functions for action nodes, typically issuing orders. Each function should only take the game state as a parameter.

*checks.py* - Contains your functions for state-based conditional checks. As with actions, each function should only take the game state as a parameter.

*run.py* - **This is the primary point of execution for your code.** show\_match runs a match between two bots then replays the match in an interface (so you can watch!). test runs a match and reports the result in the terminal.

**Notes on construction a behavior tree:**

* 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.

Example construction of a behavior tree:

# The root of your tree. Can be a selector or sequence.

root = Selector(name='High Level Ordering of Strategies')

# A sequence containing an offensive strategy

offensive\_plan = Sequence(name='Offensive Strategy')

largest\_fleet\_check = Check(have\_largest\_fleet)

# have\_largest\_fleet is a function from checks.py

attack = Action(attack\_weakest\_enemy\_planet)

# attack\_weakest\_enemy\_planet is a function from behaviors.py

offensive\_plan.child\_nodes = [largest\_fleet\_check, attack]

#this sets the sequence’s child nodes

# Similarly, this sequence checks if a neutral planet is available,

# and if so, attempts to capture one.

spread\_sequence = Sequence(name='Spread Strategy')

neutral\_planet\_check = Check(if\_neutral\_planet\_available)

spread\_action = Action(spread\_to\_weakest\_neutral\_planet)

spread\_sequence.child\_nodes = [neutral\_planet\_check, spread\_action]

# Once the various strategies have been constructed, they are hooked into the root.

root.child\_nodes = [offensive\_plan, spread\_sequence, attack.copy()]

print(root.tree\_to\_string()) # this outputs the behavior tree in string form

**Evaluation**

Your goal is to design a reactive bot with a single behavior tree which successfully completes (wins against the opponent in) a series of test environments, each representing a unique challenge. The test cases are set up in *run.py* such that you may test your bot.

Extra Credit: 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.

**Grading Criteria** (equal weighting for each item)

* Your bot operates within the time constraint of 1 second per turn.
* Your bot completes each test case (individual points per test case).
* Submission of your bot implementation as well as a text file of the output from print(root.tree\_to\_string()).

**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](https://docs.python.org/2/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.