Spike: Spike Project
Title: Tactical Analysis with PlanetWars

Author: Tran Duc Anh Dang | 103995439

Goals / deliverables:
A Spike is an agile software engineering practice used to overcome some gap in knowledge or skills. In these cases, it is difficult to build a plan or design in which you can have much confidence. The aim of the spike is to overcome these issues as quickly as possible.

Technologies, Tools, and Resources used:
List of information needed by someone trying to reproduce this work

- Python 3+
- Built in Python libraries.
- IDE or Code Editor (Visual Studio Code)

Tasks undertaken:

- Install Python: Download and Install Python 3+ via https://www.python.org/downloads/
- Set up a code editor or IDE: Download and install a python compatible ide or code editor such as Visual Studio Code, PyCharm
- Open and familiarize with the code by reading through, paying attention to the comments that had been made.
- Run the code: Execute the code and observing the output.

What we found out:
In this BlankoEXTW5, this is an improvement from the previous week representing A.I strategy for PlanetWars game. Blanko has follows a set of rules to attack, defend, and scout.

- Tactical analysis: Blanko considers the vulnerability and strategic importance of planets when attacking and defending.
- Fog Of War: Blank manages incomplete or incorrect information by tracking the last seen state of enemy plannets and updating its knowledge with new scouting information.
- Symmetrical and asymmetrical maps: Blank detects symmetry and adjust its based on map layout, exploiting choke points or priority some high value planets. However, this method hasn't been applying to Blanko's strategy yet.
- Programming structured: Blanko has been re-structured from its previous code, therefore it looks more organized as well as easier to read/debug.

Code for Blanko Week 4:

```
class BlankoEXT:

    def __init__(self):
        self.target_planets = set()

    def update(self, gameinfo):
        # only send one fleet at a time
        if gameinfo.my_fleets:
            return

        # check if we should attack or defend
        if gameinfo.my_planets and gameinfo.not_my_planets:
            # Always send from the planet with the highest value
            src = max(gameinfo.my_planets.values(), key=lambda p: p.num_ships)

            # Filter planets based on distance and ship count
            max_distance = 100
            less_ships = filter(lambda x: x.num_ships < round(src.num_ships * 0.75) and x.id not in self.target_planets and src.distance_to(x) <= max_distance, gameinfo.not_my_planets.values())

            # Choose destination based on the highest value that represents the ratio between distance, value, and growth rate, prioritizing weaker planets
            dest = max(less_ships, default=min(gameinfo.not_my_planets.values(), key=lambda p: p.distance_to(src)), key=lambda p: (2 * p.num_ships + p.growth_rate) / p.distance_to(src))

            # launch new fleet if there's enough ships
            if src.num_ships > 10:
                gameinfo.planet_order(src, dest, int(src.num_ships * 0.75))
                self.target_planets.add(dest.id)

            print("Planet {} attacked Planet {} from a distance of {:.2f} with {} ships".format(src.id, dest.id, round(src.distance_to(dest)), round(src.num_ships * 0.75)))



            # Defend planets under attack
            #Checking if there's any incoming fleets?
            for planet in gameinfo.my_planets.values():

                incoming_fleets = []
```

```
            for fleets in gameinfo.enemy_fleets:
                fleet = gameinfo.get_fleet_by_id(fleets)

                #print(fleet)  # Testing

                # if fleet.dest.id == planet.id:
                #         incoming_fleets.append(fleet)



                try:
                    if fleet.dest.id == planet.id:
                        incoming_fleets.append(fleet)
                except AttributeError as e:
                    print(f"AttributeError: {e}")

            # If there is, send fleets for defends or find friendly planet for reinforcement
            if incoming_fleets:
                total_incoming_ships = sum(fleet.num_ships for fleet in incoming_fleets)
                if planet.num_ships < total_incoming_ships:
                    # find the nearest friendly planet with enough ships for reinforcement
                    planets_with_enough_ships = [p for p in gameinfo.my_planets.values() if p.num_ships >
total_incoming_ships and p.id != planet.id]
                    nearest_planet = min(planets_with_enough_ships, default=None, key=lambda p: p.distance_to(planet))

                    if nearest_planet:
                        required_ships = total_incoming_ships - planet.num_ships + 1
                        gameinfo.planet_order(nearest_planet, planet, required_ships) # Send reinforcement

                        print("Planet {} sent {} ships to defend Planet {}".format(nearest_planet.id, required_ships, planet.id))



        # Send scouts to enemy planets
        for planet in gameinfo.enemy_planets.values():
            if planet.id not in self.target_planets:
                scout_src = min(gameinfo.my_planets.values(), key=lambda p: p.distance_to(planet))
                gameinfo.planet_order(scout_src, planet, 1) # Send scouts
                self.target_planets.add(planet.id)
```

Code for Blanko Week 5

```
class BlankoEXTW5:

    def __init__(self):
        self.target_planets = set()
        self.last_seen = {}
        self.is_symetrical = None
        self.turn = 0

    def detect_map_symmetry(self, gameinfo):
        planet_coords = [tuple(p.position) for p in gameinfo.planets.values()]
        planet_coords_reversed = [(x[1], x[0]) for x in planet_coords]
        return set(planet_coords) == set(planet_coords_reversed)


    def get_least_recently_seen(self, gameinfo):
        if not self.last_seen:
            return None
        min_turn = min(self.last_seen.values())
        planet_id = [k for k, v in self.last_seen.items() if v == min_turn][0]
        return gameinfo.get_planet_by_id(planet_id)

    def get_most_recently_seen(self, gameinfo):
        if not self.last_seen:
            return None
        min_turn = max(self.last_seen.values())
        planet_id = [k for k, v in self.last_seen.items() if v == min_turn][0]
        return gameinfo.get_planet_by_id(planet_id)

    def attack(self, gameinfo):
        if gameinfo.my_planets and gameinfo.not_my_planets:
            # Always send from the planet with the highest value
            src = max(gameinfo.my_planets.values(), key=lambda p: p.num_ships)

            # Filter planets based on distance and ship count
            max_distance = 100
            less_ships = filter(lambda x: x.num_ships < round(src.num_ships * 0.75)
                                and x.id not in self.target_planets
                                and src.distance_to(x) <= max_distance,
```

```
                        gameinfo.not_my_planets.values())

        # dest = max(less_ships,
        #            default=min(gameinfo.not_my_planets.values(),
        #            key=lambda p: p.distance_to(src)),
        #            key=lambda p: (2 * p.num_ships + p.growth_rate) / p.distance_to(src))

        # Choose the most recently seen enemy planet as the destination
        dest = self.get_most_recently_seen(gameinfo)

        if dest is not None:
            # Choose destination based on the highest value that represents the ratio between distance, value, and growth rate, prioritizing weaker planets
            dest = max(less_ships,
                       default=min(gameinfo.not_my_planets.values(),
                       key=lambda p: p.distance_to(src)),
                       key=lambda p: (2 * p.num_ships + p.growth_rate) / p.distance_to(src))



        # launch new fleet if there's enough ships
        if src.num_ships > 10:
            gameinfo.planet_order(src, dest, int(src.num_ships * 0.75))
            self.target_planets.add(dest.id)

        print("Planet {} attacked Planet {} from a distance of {:.2f} with {} ships".format(src.id, dest.id, round(src.distance_to(dest)), round(src.num_ships * 0.75)))

    def defend(self, gameinfo):
        for planet in gameinfo.my_planets.values():
            incoming_fleets = []

            for fleets in gameinfo.enemy_fleets:
                fleet = gameinfo.get_fleet_by_id(fleets)

                #print(fleet)  # Testing

                # if fleet.dest.id == planet.id:
                #         incoming_fleets.append(fleet)
```

```
            try:
                if fleet.dest.id == planet.id:
                    incoming_fleets.append(fleet)
            except AttributeError as e:
                print(f"AttributeError: {e}")

            # If there is, send fleets for defense or find friendly planet for reinforcement
            if incoming_fleets:
                total_incoming_ships = sum(fleet.num_ships for fleet in incoming_fleets)
                if planet.num_ships < total_incoming_ships:
                    # find the nearest friendly planet with enough ships for reinforcement
                    planets_with_enough_ships = [p for p in gameinfo.my_planets.values() if
                                                 p.num_ships > total_incoming_ships and p.id != planet.id]
                    nearest_planet = min(planets_with_enough_ships, default=None, key=lambda p: p.distance_to(planet))

                    if nearest_planet:
                        required_ships = total_incoming_ships - planet.num_ships + 1
                        gameinfo.planet_order(nearest_planet, planet, required_ships)  # Send reinforcement

                        print("Planet {} sent {} ships to defend Planet {}".format(nearest_planet.id, required_ships, planet.id))


    def send_scouts(self, gameinfo):
        for planet in gameinfo.enemy_planets.values():
            if planet.id not in self.last_seen and planet.id not in self.target_planets:
                scout_src = min(gameinfo.my_planets.values(), key=lambda p: p.distance_to(planet))
                gameinfo.planet_order(scout_src, planet, 1)  # Send scouts (1 ship)
                self.target_planets.add(planet.id)
                #self.last_seen.add(planet.id)
                self.last_seen[planet.id] = self.turn
                print("Planet {} sent {} ships to scout enemy Planet {}".format(scout_src.id, 1, planet.id))


    def update(self, gameinfo):
        #print("Turn Numbers:", self.turn)
        self.turn += 1

        # Send scouts to unexplored enemy planets
        self.send_scouts(gameinfo)
```

```
# Defend planets under attack
self.defend(gameinfo)

# Only send one fleet at a time
if gameinfo.my_fleets:
    return

# Attack enemy planets
self.attack(gameinfo)
```

Here are some main differences between 2 versions:

Additional attributes:

- self.last_seen
- self.is_symmetrical
- self.turn

Additional methods:

- detect_map_symmetryc
- get_least_recently_seen
- get_most_recently_seen

Game logic has been separate into multiple methods making it more modular (attack, defend, send_scouts).

Overall, in the newest version, Blanko has been more orgranized and modular with improved logic for attack, defend and send scouts, it also adds functionality to keep track of the last seen enemy planets and detect map symmetry which will be applying in the future.