**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.