Spike Summary Report

💡 **Spike:** Task_07
Title: Tactical Analysis with PlanetWars
Author: Thomas Horsley, 103071494

Goals & Deliverables

Aim: The overall goal of the spike was to develop the skills necessary in writing a PlanetWars bot who's capable of using tactical analysis to inform it's decisions in the context of a strategic game model.

Deliverables:

By date of submission, the following must be included:

- Agent code demonstrating simple use of tactical analysis
- A baseline (no tactical analysis) bot for performance benchmarks
- A submission report containing tactical-analysis-bot performance benchmark data

Technology, Tools and Resources

💡 The project was scripted in Python 3.10.9 using the VSCode IDE Version 1.76.
Source control is handled using Git with the BitBucket GUI and UMLs and diagrams are built at *www.lucidchart.com*
Optionally (though recommended), are the VSCode extensions listed below.
Packages included in the standard library such as *math* and *random* will not be included in this list.

VSCode Plugins/Extensions

- Python
    - Author: Microsoft
    - Version: 2023.4.1
- Pylance
    - Author: Microsoft
    - Version: 2023.3.20
- Colorful Comments
    - Author: Parth Rastogi
    - Version: 1.0
- Code Runner
    - Author: Jun Han
    - Version: 0.12.0
- Doxygen Document Generator
    - Author: Christoph Schlosser
    - Version: 1.4.0
- Todo Tree
    - Author: Gruntfuggly
    - Version: 0.0.224

Resources

- Echo 360 Lecture set *"Topic 03 - Graphs, Strategies and Tactics"*
- Echo 360 Lecture *"Topic 02.2 - Game Balance & Bias"*
- Modules, *"Topic - Rules, Balance & Goals"*
    - Author: Clinton Woodward

Relevant Packages

- Pyglet
    - Author: **pyglet**
    - Version: 2.0.5
- Openpyxl
    - Author: Eric Gazoni, Charlie Clark
    - Version: 3.1.2

Tasks Undertaken

Tactical Analysis Bot

> My bot’s overall strategy was to take control of the closest available planets with the most resources. This resulting in the bot valuing strong control over dense clusters of planets as-well as rapid initial growth (not wasting any time on more advanced inputs such as scouting).

The bot had access to a game facade (player-relevant, publically known frame-data relative to the current gamestate) and from this facade, the bot may scrape data relevant to its’ planets, visible opponents planets, the fleet count of planets and the growth of planets. The bot will use this information to gauge the most strategically “valuable” (valuable relative to the bots strategy) positions before making efforts to capture those positions.

```
class FIBD:
    # Summary: If this planet is an enemy then check if we can overwhelm with
    # our planets fleet, if so then attack with force.
    def update(self, gameinfo):···

    # Summary: Determine the distance between source planet and unowned planets
    def determinePlanetDistances(self, src, otherPlanets):···

    # Summary: Take the planet distances, chose the planet associated with the min
    # distance and return it
    def findClosestPlanet(self, src, otherPlanets):···

    # Summary: Determine the size of planets fleet and attack with overwhelming
    # force
    def determineFleetSize(self, src, dest):···

```

All the functionality and gameinfo the bot (FIBD) needs in order to employ strategy

Each update cycle the bot orders it’s largest planet to send a fleet to the closest neutral or enemy planet. As FIBD employs tactical analysis to make it’s decisions, it can determine if it’s possible to take a destination planet by force or not. If this isn’t possible, the bot will attempt to prevent further growth of that planet by sending smaller fleets until it’s fleets are large enough to take the resource.

```python
    # Summary: Determine the distance between source planet and unowned planets
    def determinePlanetDistances(self, src, otherPlanets):
        distList = []
        # ? Calc dist between source planet and other planet
        for planet in otherPlanets:
            distList.append(
                [planet, sqrt((planet.x - src.x) ** 2 + (planet.y - src.y) ** 2)]
            )

        return distList

    # Summary: Take the planet distances, chose the planet associated with the min
    # distance and return it
    def findClosestPlanet(self, src, otherPlanets):
        dest = (None, 200)  # ? Initial value
        validPlanets = []

        for subset in self.determinePlanetDistances(src, otherPlanets):
            if subset[1] <= src.vision_range:  # ? Compare distances
                validPlanets.append(subset[0])

        if not validPlanets:
            for subset in self.determinePlanetDistances(src, otherPlanets):
                if subset[1] <= dest[1]:  # ? Compare distances
                    validPlanets.append(subset[0])

        return choice(validPlanets)
```

How the bot calculates the set of closest planets.

Each update tick the bot will find the unowned planets before calculating their distances from the fleet source. This data is appended to a 2-dimensional array which is passed to a comparitive function that returns the closest planet from the set of valid planets. In the case that there's multiple, equally attractive destination planets, the bot will randomly chose one.

The looping and iterative nature of these functions caused of UPS bottlenecking on larger map files. To solve this in future iterations, the AI will store already known information and modify this information relative to the current gamestate (more akin to memory) rather than build this information each update cycle.

Modification to Main.py

💡 To gather a non-trivial amount of performance statistics, it was important that the bot played a minimum of 150 matches on a total of 15 maps. This was achieved through modification of the *main( )* function from *main.py*

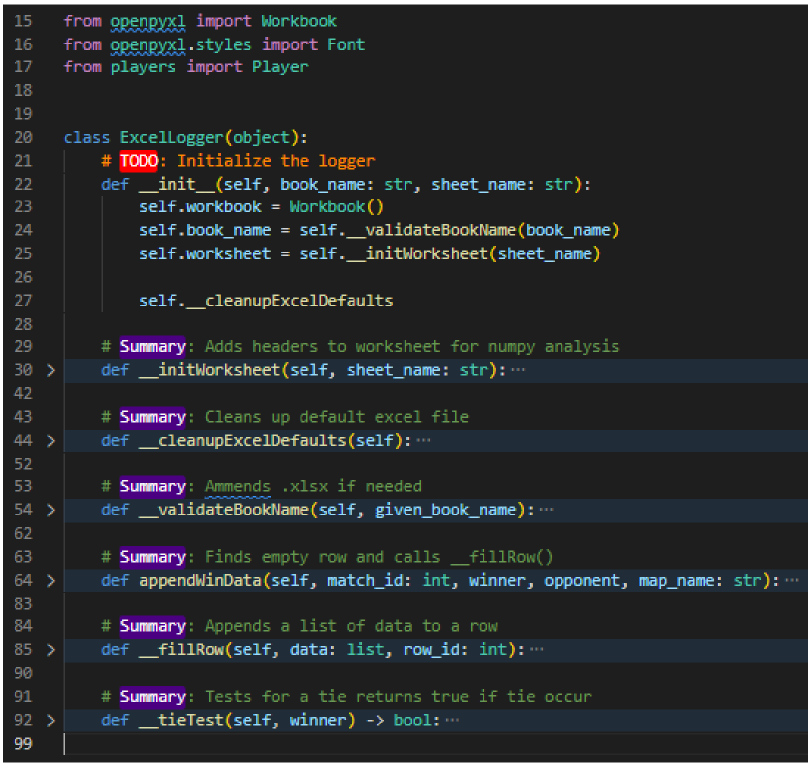
The function was modified with a simple loop to allow the Excel Logger class to be called each time a game window closed. This allowed for settings modification during runtime which meant the logger could record multiple games on multiple maps within a single run.

```python
if __name__ == "__main__":
    # Supplied map sizes were generated for (500, 500) size
    # Scale the visual size with this
    screen_size_scale = 2
    current_map = 10
    current_match = 100
    matches_per_map = 10
    max_matches = 150

    first_map = True
    excel_logger = ExcelLogger(
        "PlanetWars4", f"Win Data Matches {current_match} - {max_matches - 1}"
    )

    while current_match < max_matches:
        if current_match % matches_per_map == 0 and not first_map:
            current_map += 1

        map_file = f"./maps/map{current_map}.txt"

        settings = {
            "map_file": map_file,
            "players": ["FIBD", "Rando"],
            "max_game_length": 600,
            "start_paused": False,
            "game_over_quit": True,
            "update_target": 60,
            "background_img": False,
            "width": int(500 * screen_size_scale),
            "height": int(500 * screen_size_scale),
            "vsync": False,
            "resizable": False,
        }

        window = PlanetWarsWindow(**settings)
        app.run()
        window.game.logger.flush()

        excel_logger.appendWinData(
            current_match,
            window.game.winner,
            [p for p in window.game.players.values() if not p.is_alive()],
            f"map {current_map}",
        )

        current_match += 1
        if first_map:
            first_map = False
```

During runtime, the performance of the program waned relative to the current loop iteration. This would result in an initial *update_rate* of 200Ups being capped at 5Ups not 10 matches later. This bug has never resulted in a program crash and such the cause remains unknown.

The Excel Logger

💡 The Excel Logger tool was made using the openpyxl package and allowed automation of exporting PlanetWars win data to .xlsx format. This class was written to be scalable and outputs array data readable by numpy if deeper analysis is needed.



```python
from openpyxl import Workbook
from openpyxl.styles import Font
from players import Player


class ExcelLogger(object):
    # TODO: Initialize the logger
    def __init__(self, book_name: str, sheet_name: str):
        self.workbook = Workbook()
        self.book_name = self.__validateBookName(book_name)
        self.worksheet = self.__initWorksheet(sheet_name)

        self.__cleanupExcelDefaults

    # Summary: Adds headers to worksheet for numpy analysis
    def __initWorksheet(self, sheet_name: str):···

    # Summary: Cleans up default excel file
    def __cleanupExcelDefaults(self):···

    # Summary: Ammends .xlsx if needed
    def __validateBookName(self, given_book_name):···

    # Summary: Finds empty row and calls __fillRow()
    def appendWinData(self, match_id: int, winner, opponent, map_name: str):···

    # Summary: Appends a list of data to a row
    def __fillRow(self, data: list, row_id: int):···

    # Summary: Tests for a tie returns true if tie occur
    def __tieTest(self, winner) -> bool:···
```

ExcelLogger class function overview.

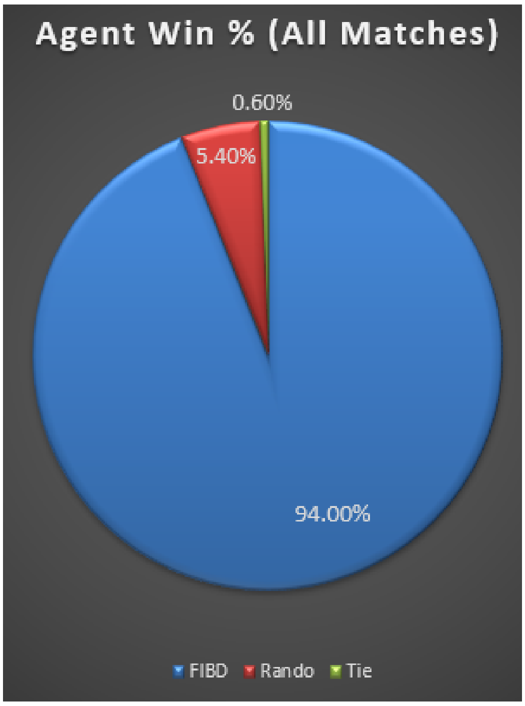
This class will initialize and empty a new excel workbook, build sheets with headers and style the header cells.

The method *appendWinData( )* is called each time a bot match ends. This function finds an empty row in the sheet, appends [*match id*, *winner_name*, *loser_name*, *map_played*].

Additionally, the class is able to deal with tie situations (as it wouldn't be passed a *winner_name* or *loser_name* string value).

Outcomes

💡 FIBD played Rando 150 times on 15 seperate maps over the course of 4 hours. The results of these matches are as follows.



Out of 150 matches played FIBD won 141 (94.0% win rate) against an opponent who randomly chose their move. Whilst the results provied an insightful baseline for agent performance, they weren't unexpected as even a trivial level of strategic reasoning should beat a random moveset.

More interestingly, the bot struggled on smalled maps with fewer resources, this is most noticible in *FIBD Win %* on map0. This occured as FIBD's strategy wasn't suitable for the map being played as fewer planets allows more opportunities for a tactically random agent to make best moves. Additionally, a tie only occured once throughout the 150 matches and was the result of match time ticking to zero.

Map (10 Matches per)	FIBD Win %	Rando Win %	Tie %
0	60%	30%	10%
1	100%	0%	0%
2	100%	0%	0%
3	90%	10%	0%
4	100%	0%	0%
5	100%	0%	0%
6	100%	0%	0%
7	90%	10%	0%
8	100%	0%	0%
9	80%	20%	0%
10	90%	10%	0%
11	100%	0%	0%
12	100%	0%	0%
13	100%	0%	0%
14	100%	0%	0%