# CSCI 1933 Project 4: PlanetWars

Due date: 12/06/2017, before midnight

## Introduction

Up to this point in the semester, you've had a fairly specific format for your projects, with skeleton code that we've given you and specific methods that we've asked you to implement. This project is going to be a bit different. This time, your job is to implement a strategy for a game. You have quite a bit of flexibility in how you implement the strategy; we just have a specific way to call your strategy in the end. But more details on that later; first let's go over the game rules and how it works.

## The Game

### Premise

PlanetWars is a game where two players are pitted against each other in a system of interconnected planets. The players each start out on a home planet and the object of the game is to expand and eventually take over all of the opponent's planets.

### Game Board

The game board is set up as a graph of planets interconnected by weighted edges.

- Population growth occurs at a set rate on each planet.
- If two planets are connected by an edge, then shuttles can be sent between these planets. The distance between the planets affects how many turns it takes for shuttles to move from one planet to the other.
- The planets vary by two factors:
    - Size correlates to the total population a planet can support. Once a planet hits its maximum population, population growth will cease and any population that exceeds the maximum (for example, from incoming shuttles) will decrease by the rate described below.
    - Habitability correlates to the population growth rate on a planet. Population change after one turn for a given planet is defined below.
        - Let $c$ = current population, $m$ = max population, $g$ = growth rate, and $p$ = overpopulation penalty.
            - If $c < m$, pop next turn = $c*g$
            - If $c >= m$, pop next turn = $c - (c-m)*p$
        - Currently, $p = 0.1$ and $g = 1 + (\text{habitability} / 100)$

- Each player starts out with one planet with a given population. A player can send shuttles with population to neighboring planets on his/her turn.
- Each player receives information about the entire game board, including the planet IDs of all planets and which planets are interconnected. However, a player can only see detailed information (owner, population, size, habitability, incoming shuttles) about the planets that they own and their neighboring planets.

# Game Flow

In one turn, a player

- Receives information about the game state
- Adds moves to the event queue
    - A ‘move’ is sending a shuttle from one owned planet to a neighboring planet. The player can set how much population is sent in the shuttle.
    - A player can make as many moves as he/she wants in a turn.
- Returns the event queue to the game engine

The game engine will then make all legal moves in the event queue, and allow a half-step of time pass, in which

- Population growth (or decay, if overpopulation cap) occurs on all planets
- All shuttles move half a step

The game play then passes to the other player.

# Shuttles and Landings

A shuttle carries some amount of population from Planet A to Planet B. Let’s say that Player 1 sent the shuttle with population 100, and the distance between the planets is 2. The shuttle takes two full turn cycles to arrive at Planet B, and during that time, no population growth occurs on the shuttle. There are several possibilities when the shuttle arrives at Planet B.

## Player 1 Owns Planet B

All population belongs to the same player, so the population of the shuttle is simply added to the total population of the planet.

## Player 2 Owns Planet B

A battle commences. The defending population has a 10% advantage, so attacking a planet with the same number of people as are currently on it will not succeed. The attackers need 110% of the planet population to conquer the planet, and very small attacks will have no effect on the population of the planet. The outcome is determined by the following rules:

- If *1.1 * oldPlanetPop > attackingPop*
    - *newPlanetPop = min(oldPlanetPop, 1.1* oldPlanetPop - attackingPop)*
    - Owner does not change
- If *1.1 * oldPlanetPop = attackingPop*
    - *newPlanetPop = attackingPop - 1.1 * oldPlanetPop*

    - Owner changes to the attacking player
- If *1.1 * oldPlanetPop = attackingPop*
    - *newPlanetPop = 0*
    - Neither player now owns the planet

Whichever side wins the battle, wins the planet, and stays there with the remaining population. In the unlikely circumstances that both sides lose all of their population at the planet, it becomes neutral and is up for grabs for whoever gets a shuttle there first.

### Planet B is Neutral

Player 1 captures the planet, and the attacking force lands successfully. No population is lost.

Shuttles are processed in an order which rewards both defensive assistance and multiple concurrent attacks. All friendly shuttles which are scheduled to arrive on a turn are landed, and their population is added to the planet's population. Following this, the attacks of all hostile shuttles scheduled to arrive are coalesced into one larger attack - if two hostile shuttles arrive at the same time, one with population 10 and one with population 15, then the attacking force used to compute a winner has population 25.

# Your Task

Write a class that implements `iStrategy`. As you will see when you look at `iStrategy`, there is one required method that needs to be written. (Of course, we strongly recommend that you do **not** include all of your logic in one method, due to various reasons that we have discussed throughout the semester.) At each turn, the `takeTurn` method in your strategy class will be called with parameters that include a list of the planets in the system and a queue for you to add your moves to. You will use the information in the list of planets and other information that you get from interacting with the system to make intelligent decisions about what moves to add to the queue.

We have provided you with a very simple strategy to give you an idea of how to interact with the game engine, add moves to the queue, and access information about the game space.

There are many different possible strategies, so each person/team's approach may look very different. However, we do want each of your strategies to satisfy a few basic specifications:

- You must use the move queue properly and add moves at some point throughout each game.
- Not counting the move queue described above, you must use at least **three** of the following abstract data types in reasonable ways to help in implementing your strategy: lists, queues, dictionaries (maps), stacks, or graphs. Feel free to use Java built-in data structures that implement these ADTs (e.g. for a dictionary, you can use the HashMap class). NOTE: we are not requiring you to implement your own data structures for these.

- Your strategy must attempt traversal of the graph in some form. What exact traversal you use or what data you use in deciding how to traverse the planet graph is flexible, but your strategy should involve some attempt to traverse the graph.

In addition to the `takeTurn` method, you will need to implement two methods in your Strategy class that help us identify your submission:

`public String getName( )`: this method will be used as your team name on any scoreboards we report.

`public boolean compete( ):` return "true" in this method if you want your submission to compete in the class tournament, "false" if not.

# Grading

Due to the unique nature of this project, you will be graded in a different manner than previously. Your strategy will be tested against several strategies that we have created. You will be graded based on performance against these strategies as well as on your proper use of data structures and the style of your implementation. We will provide you with these strategies in advance so you have a benchmark for how your strategy actually performs.

Grading components:

- Performance against our example strategies: 50 pts
    - If your solution can beat our RandomStrategy in > 70% of the trials (out of 100 random runs): 30/50 points
    - We will provide at least three additional non-random strategies (AIStrategy1, AIStrategy2, …). For each additional strategy you can beat in > 70% of the trials (out of 100 random runs), you will earn an additional 10 pts toward the 50 total (a maximum of 50 pts can be earned).

- Appropriate use of requested data structures (at least **three** of lists, queues, dictionaries, stacks, or graphs, not counting the move queue): 30 pts

- Overall design and style of your strategy implementation: 20 pts

# Running the Game

This example assumes that you're using IntelliJ as your Java IDE.

1. Download the assignment from the class moodle and unzip the folder
2. Import the folder "project4" as a project into IntelliJ
3. On the libraries screen, uncheck the "strategies" folder
    - This can be done later if necessary via File -> Project Structure -> Libraries

4. To add a strategy of your own, create a new java file in the folder project4/src/planetwars/strategies
    - Use the provided example strategies as a reference
    - Note that you only need to edit the takeTurn method
5. To run the game WITH graphics...
    - Open the file project4/src/planetwars/publicapi/Driver.Java
    - Edit the file to call your strategy instead of the provided strategies
        - For a strategy at project4/src/planetwars/strategies/MyStrategy.java…
        - Create a .jar file for your strategy by going to File -> Project Structure -> Project Settings -> Artifacts -> Plus sign -> Jar -> From modules with dependencies
        - Name your strategy and set its output folder to project4/strategies
        - Choose any of the available classes as the main class
        - Press ok, go to the menu Build -> Build Artifacts -> Build
        - In the main method, change the window to run your strategy
            - E.g. GameWindow window = new GameWindow(MyStrategy.class, RandomStrategy.class);
6. To run the game WITHOUT graphics (this can be useful for testing your strategy)...
    - Open the file project4/src/planetwars/core/PlanetWars.Java
    - Edit the file to call your strategy instead of the provided strategies
        - For a strategy at project4/src/planetwars/strategies/MyStrategy.java…
        - Import your strategy with import planetwars.strategies.MyStrategy
        - In the main method, change strategy1 to be an instance of MyStrategy
            - E.g. with IStrategy strategy1 = new MyStrategy();

# Public API

The API (interfaces you will be working are listed below) - -

1. IEdge - Interface denoting an edge between two planets.
2. IEvent - An event is a wrapper around a population transfer from a source and destination planet.
3. IPlanet - A planet which has not been discovered yet.
4. IPlanetOperations - An interface denoting the events scheduling the movement of people.
5. IShuttle - An interface to represent the movement of people.
6. IStrategy - Your strategy class should implement this interface. Essentially, your strategy class should move a fixed number of people from your conquered planets to a destination planet such that you increase the number of planets you have conquered as the game progresses.
7. IVisiblePlanet - A planet which you have conquered or is adjacent to you. You are able to see a complete set of characteristics for these planets

# Submission

Please write a brief description of your strategy, including what data structures you used to implement it, in the comments of your class that implements iStrategy. Then, zip all of Java files (.java), including your Strategy class, and submit them through Moodle.

# Working with a partner

As discussed in lecture, you may work with one partner to complete this assignment (max team size = 2). If you choose to work as a team, please only turn in one copy of your assignment. At the top of your Strategy class definition, include in the comments both of your names and student IDs. In doing so, you are attesting to the fact that both of you have contributed substantially to completion of the project and that both of you understand all code that has been implemented.