Swinburne University of Technology

Implementation Guide

Deceptive PlanetWars

Author:
Michael Jensen
6153208

Supervisor:
Clinton Woodward

October 5, 2011

1 Introduction

1.1 Purpose

The purpose of this document is to define the API provided for agent implementors when building an agent to play Deceptive PlanetWars. This guide is aimed at programmers familiar with the python programming language who will be implementing a control program for a PlanetWars player.

1.2 Background

PlanetWars is a simple turn based strategy game based on Galcon[1] and the Google AI Challenge.[2] It has been modified to include rerouting fleets in flight and fog of war.

1.3 Why write a bot?

By writing a bot for PlanetWars, you'll be helping me out with my Honours (if it's still 2011 that is) and hopefully having fun exploring AI development in a simple strategy game. The techniques you implement here could be expanded and used in a commercial strategy game for fun[3] or profit.[4]

2 PlanetWars

2.1 Gameplay

PlanetWars takes place in a "system" of planets, which are statically located in a 2-dimensional space.[5] Each player begins the game controlling at least one planet with some ships on it. The player can issue commands to the planets and fleets that they control, capturing more planets and increasing their army size. Each turn, a planet under a player's control will produce more ships, with some planets producing more ships than others.

2.2 Software Structure

In order to facilitate deception, PlanetWars has a fog of war. This prevents each player from seeing their opponent's activities when they aren't nearby. To prevent players from cheating, each player is given a `facade` object each turn which can be queried for information about the game state. The information that each facade yields will represent the parts of the game that aren't covered by fog of war. For the parts of the game that are out of vision range, the facade will contain old or possible no information. This pattern is demonstrated in Figure 1.

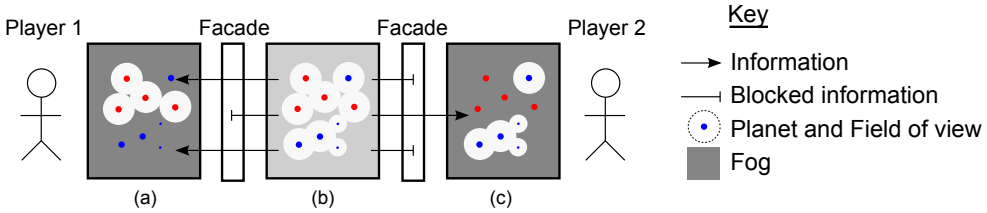

Figure 1: The facade pattern. Agents can see the worlds (a) and (c), which are filtered versions of (b).

[1] galcon.com
[2] ai-contest.com
[3] starcraftaicompetition.com
[4] http://www.google.com/search?q=jobs+game+ai
[5] Imagine that the planets are being viewed from sufficiently far away so as to appear to be on a 2-dimensional plane.

When planets are out of range, the facade will contain the last known state of that planet. Growth and control will not be visible through the facade, but the planet will still be there. When a fleet moves out of range, the facade forgets about it. Fleets that are accessible through the facade object are guaranteed to be within vision range, and fleets that are out of vision are guaranteed to not be accessible through the facade.

2.3 Game flow

The PlanetWars game loop is quite simple. Each turn, the engine updates each player's facade with the information that's in vision range. Each player's `DoTurn` method is called and passed a PlanetWarsProxy - their facade. The player can call `IssueOrder` on their facade to set an order to be executed this turn. There is no limit on the number of times you can call `IssueOrder`, except that you must send at least one ship in each order and you only have limited ships on the planets you control. Note that players may also choose not to issue any orders at all.

Once turns have been made by both players, the game engine processes them and advances the game to the next time step.

The game will continue until one player has no ships left or the time limit[6] expires, whichever comes first. The winner is the player with more ships left at the end of the game.

2.4 Classes

The simplified UML class hierarchy for PlanetWars is shown in Figure 2. As a bot implementor, you'll be mainly interacting with the `PlanetWarsProxy` class, which you can query for information about the game world. Figure 3 shows the full interface for this class. You will need to write a class that implements the `DoTurn` method. This method must take one argument, the `PlanetWarsProxy` that contains your facade and your interface to issue orders. More information about the requirements of player classes can be found in Section 3.1.

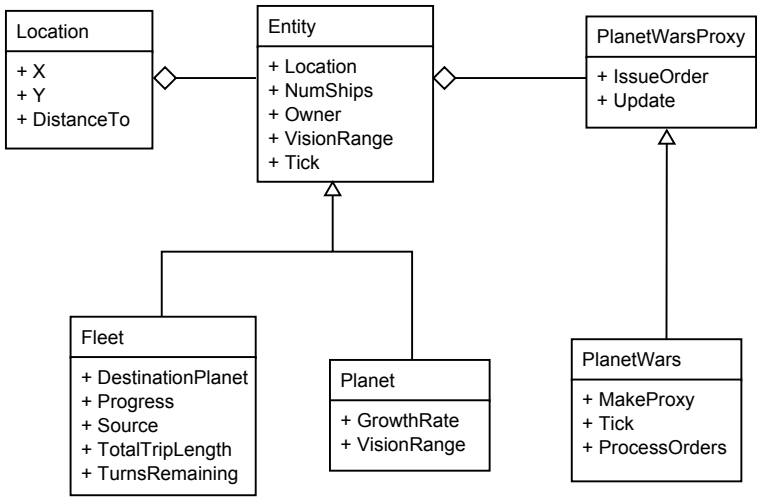

Figure 2: Simplified UML diagram of the PlanetWars classes

Both Planets and Fleets inherit from the `Entity` class. This class is shown in full in Figure 4. In general, players will only interact with this class indirectly, by calling methods from Planets and Fleets. The `Location`, `X` and `Y` methods all refer to the same location, with `X` and `Y` being convenience

[6]Currently set to 500 turns, but subject to change

methods that access the entity's location. `DistanceTo` allows players to easily calculate the distance between two entities. Note that `DistanceTo` will always return an integer, being the rounded up geometric distance. The distance is also the total number of turns it would take a fleet to travel from one entity to another. `IsInVision` and `VisionAge` allows players to tell if the entity they are querying is up to date. Players can call `VisionAge` to get the number of turns since this entity was in vision. It is automatically incremented between turns. `IsInVision` returns `True` if the entity in question is within the vision range of a friendly entity this turn, that is, if `VisionAge` is zero.

Figure 3: Complete listing of public PlanetWarsProxy interface

In addition to the methods provided by `Entity`, `Planets` have a `GrowthRate` which returns the number of ships generated by this planet each turn. Players may wish to prioritise planets with a high growth rate when choosing how to expand, as this will yield a larger army over time.

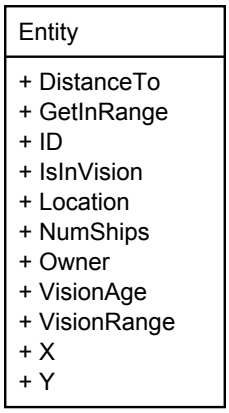

Figure 4: Complete listing of public Entity interface

Finally, the `Fleet` class provides several methods for querying a fleet in transit between two planets. The `TotalTripLength` method returns the distance between a fleet's source and destination.

`Progress` returns the number of turns that the fleet has been in flight on this leg of its journey, while `TurnsRemaining` returns the number of turns until the fleet arrives. The fleet class no longer uses a location object to represent its position, so you must use `X` and `Y` to query for a fleet's current position in the game space. `DestinationPlanet` and `Source` can be used to retrieve the endpoints of a fleet's trip. Note that since fleets can be redirected while in flight, the result of `Source` may not be a Planet object. The result of `Source` is only guaranteed to respond to the methods `X` and `Y`.

3 Making Players

3.1 Player Classes

Players in PlanetWars are required to implement a `DoTurn` method. This method will be called each turn by the game engine to give players a chance to issue orders. This method must take a single object parameter. It will be passed a PlanetWars object which contains a version of the game state, filtered for your player program. During your turn, you can query the PlanetWars object for information about the game, log messages to a file (useful for debugging/analysis) and issue orders to the fleets and planets under your control. This object's full interface is shown in Figure 3 and described in Section 3.2.1.

Your player class should extend from the `BasePlayer` class. This class has a basic constructor that assigns an ID to the player class. If you require some code to be run when your player is instantiated, make sure you call the BasePlayer's constructor as shown in Code Listing 1.

```
1 class MyPlayer(BasePlayer):
2   def __init__(self):
3     super(MyPlayer, self).__init__()
4     #your constructor code here
5
6   def DoTurn(self, pw):
7     #your player logic here
8     pass
```

Listing 1: Calling the parent class' constructor.

3.2 Class Reference

3.2.1 PlanetWarsProxy

See Table 1.

3.2.2 Entity

See Table 2.

3.2.3 Fleet

Note: Fleet inherits from Entity. This list only shows the methods added by the Fleet class.
See Table 3.

3.2.4 Planet

Note: Planets inherits from Entity. This list only shows the methods added by the Planet class.
See Table 4.

`CurrentTick`	Returns an `integer`, the current turn number. The first turn is turn 0.
`Distance`	Takes two entities and returns the distance between them. This distance is the number of turns it would take a fleet to travel between the two entities.
`EnemyFleets`	Returns a list of visible enemy fleets.
`EnemyPlanets`	Returns a list of known enemy planets.
`Fleets`	Returns a list of all visible fleets.
`GetFleet`	Takes a fleet ID and returns the fleet with that ID, if it still exists and is visible. If not, this method returns `None`.
`GetPlanet`	Takes a planet ID and returns the planet with that ID.
`IssueOrder`	Used to send ships from a planet or fleet under your control to a target planet. Takes a fleet, fleet ID, planet or planet ID as the source of ships, a planet as the destination and an integer number of ships to send.
`log`	Outputs a message to a file. This could be useful for debugging your program or tracking how it "thinks" during a game.
`MyFleets`	Returns the list of fleets under your control.
`MyPlanets`	Returns the list of planets under your control.
`NeutralPlanets`	Returns a list of planets that were neutral last time you saw them.
`NotMyPlanets`	Returns the list of planets that are not under your control. This is the union between `EnemyPlanets` and `NeutralPlanets`.
`NumFleets`	Returns the total number of known fleets.
`NumPlanets`	Returns the total number of planets.
`Planets`	Returns the list of all planets, in their last known state.
`TotalShips`	Returns the total number of ships under your control. That is, the sum of ships in all of your fleets and planets.

Table 1: PlanetWarsProxy Class interface

`DistanceTo`	Takes another Entity and returns the distance between them. Like `PlanetWars.Distance`, returns an integer being the number of turns it would take a ship to travel from this entity to the other.
`GetInRange`	Takes a list of entities and returns a dict containing entities from the list that are within vision range of this entity. The dict's format is {id:entity}.
`ID`	Returns the ID of this entity.
`IsInVision`	Returns true if this entity is within vision range this turn.
`NumShips`	Returns the number of ships on or in this entity.
`Owner`	Returns a number representing the owner of this entity, as a string. Neutral entities are owned by player "0".
`VisionAge`	Returns the number of turns since this entity was seen. Entities that are within vision range this turn will have a `VisionAge` of 0.
`VisionRange`	Returns the vision range of this entity. The units used here are consistent with those returned from `DistanctTo`.
`X`	Returns the x component of this entity's location.
`Y`	Returns the y component of this entity's location.

Table 2: Entity Class interface

`DestinationPlanet`	Gets and returns the ID of the planet that this fleet is travelling towards.
`Progress`	Returns the number of turns that this fleet has been travelling.
`Source`	Returns the location that this fleet is travelling from.
`TotalTripLength`	Returns the total number of turns required for the fleet to arrive at its destination.
`TurnsRemaining`	Returns the number of turns remaining before the fleet arrives at the destination. Note that this will never reach 0. This is, on the turn before the fleet arrives, `TurnsRemaining` will return 1.

Table 3: Fleet Class interface

`GrowthRate`	Returns the number of ships that a generated by this planet each turn. As long as this planet is not neutral, it will start each turn with `GrowthRate` additional ships.
`Location`	A convenience method that retreives the location of a planet. You may find this more convenient that calling `X` and `Y`.

Table 4: Planet Class interface

4 Testing Your Bot

4.1 Set up

To set up your system to run PlanetWars, you need to have pygame installed. This can be obtained from *http://pygame.org* and set up on an existing python installation with very little hassle. Once installed, execute `Game.py` from the command line, passing in the path to a map file. See Listing 2.

```
> python Game.py ../maps/map1.txt
```

Listing 2: Running PlanetWars

4.2 Testing against other bots

To modify which bots are used when you run the game, open up `Game.py`. Scroll down to the `if __name__ == '__main__':` block and modify the definitions of `bot1` and `bot2`. You may also need to modify the import statements on the lines above, if you put your bot in a different package or module. Listing 3 shows a snippet of `Game.py` where you should insert your bot code.

```
if __name__ == '__main__':
  ...
  try:
    from Players.MyPlayer import MyPlayer
    from Players.ScoutPlayer import ScoutPlayer
    bot1 = MyPlayer() #your player!
    bot2 = ScoutPlayer()
    ...
```

Listing 3: Example bot loading code

5 Strategy Tips

5.1 Utilising Game Mechanics

Over the years, two important components of strategy games have been developed, macromanagement and micromanagement. Macromanagement (knows simply as "Macro") refers to the control of your forces as a whole, including your economy, military and (in some games) technology. Micromanagment (known as "Micro") is the control of individual units within your army. These concepts also relate to *Strategies* and *Tactics*. Macromanagement, much like strategies, involves actions that will generally have a long term benefit, while micromanagement is more concerned with tactical decision making and short term benifit.

In PlanetWars, the macro aspect is capturing planets to increase the size of your army. This is a good idea because a larger army means you'll almost certainly be victorious. Micro in PlanetWars is limited, but involves the turn by turn control of individual fleets.

5.2 Scouting

Due to the fog of war, information in PlanetWars is very limited. When deciding how to expand your fleets, consider how the planet's vision can affect gameplay. If your opponent is trying to hide things from you in the fog, you may spot them, block an attack path or at least have a bit more warning.

5.3 Defense

Defense in PlanetWars is just as important as attacking. When you lose a planet to your opponent, your army growth slows and his speeds up. This two-fold damage makes it important to make sure

you don't lose too many strongholds during the game. This also highlights the importance of scouting: if you only have a few turns to react, you may not be able to defend your planets.