ReAntics Student Manual

How to play ReAntics

Rules of the game

ReAntics is a turn-based tactics game which revolves around two players competing via two colonies of ants. Each player has his own anthill, queen ant and any other ants or tunnels he has built. The anthills can be used for "building" new ants and both anthills and tunnels can store food collected by ants. Each player begins the game with a single queen ant and loses the game if the queen is killed. The other ants must be bought with food that is gathered by worker ants.

Victory can be achieved either through military or economic dominance of the map. Specifically, a player will win if his opponent's queen is killed, his opponent's anthill is captured, or if the player collects 11 units of food.

The game begins with a 10x10 (grid-based) board where each player places an anthill and tunnel on their side of the board. After placement, the queen ant is placed on the anthill and a worker ant is placed on the tunnel. As depicted by the red and blue boundaries in the game, the board is divided into three parts. The top four rows are Player 1's side while the bottom four rows are Player 2's side. The middle two rows belong to neither player.

Player 1 begins the game's setup phase by placing her anthill, tunnel, and 9 grass pieces on her side of the board.. Lastly, players place two food sources on their enemy's side of the map. After this last step, the players' queen ants are placed on their respective ant hills, worker's on their respective tunnels, and the setup phase is complete.

After setup, the player designated to go first make one or all of these actions: he may move each ant once which opens up the option to attack or build one ant from his anthill. When the player is done with his turn he clicks the "End" button. Only one ant is allowed to occupy a tile at a time.

To move an ant the player will click the ant she wishes to move and its square will be highlighted in green to show its selected. Any tiles the ant is allowed to move to will be highlighted green. Clicking on a highlighted tile will move the ant to that tile and shade it grey to show that it has already moved this turn.

If the move puts the ant in attack range of one or more enemy ants, all the enemy ants in range will be highlighted in red. To attack one of the ants the player may click on it. An attack must be made when available and, strategy-wise, there is no reason not to do so.

To create an ant the player selects his empty anthill. A popup menu will appear listing the four buildable ants. The player may click the ant type he wants to create and it will be created on top of the anthill if they have enough food. Or the player may click elsewhere on the board or press the esc key to cancel building an ant. The ant that is created cannot be moved until the next turn and so is shaded grey upon creation.

Each ant type performs slightly differently in the game according to performance attributes for Movement, Health, Attack, Range, and Build Cost. The performance attributes for the five ant types are summarized in Table 1.

Table. The Ant Performance Attributes

Attribute	**Worker**	**Drone**	**Soldier**	**Ranged Soldier**	**Queen**
Special Characteristics	Can carry food	Ignores grass movement cost		Ignores grass movement cost	Results in loss if killed Cannot leave home zone

Movement: The number of tiles and ant can move in one turn.	2	3	2	1	2
Health: The amount of damage an	4	4	8	2	10
Attack: The amount of damage the ant does when it attacks	0	2	4	3	4
Range: The distance from which this ant can attack	N/A	1	1	3	1
Build Cost:The amount of food it takes to build this ant	1	2	2	2	N/A

The statistics for the four different construction types are show below. Constructions can be buildings, obstacles, or food.

Table. The Construction Type Statistics

Attribute	**Anthill**	**Tunnel**	**Grass**	**Food**
Movement Cost: The number of movement points it takes to move an ant onto this tile.	1	1	2	1
Capture Health: The number of turns an ant must be on this construction to capture it.	3	N/A	N/A	N/A

How to start a game

QuickStart

Select 1+ Agents:
1: Human v. AI
2: AI v. AI single set
3+: Round Robin between all of the agents selected

Enter a number of games to play (defaults to 1)

Click the Green **Quickstart** Button (top right).
(** Note: Anything selected in **Additional Settings** will also be applied **)

Start

Click on the dropdown menu at the top right, currently labeled “QuickStart”.

Select an option:
Two Player: select a player one and a player two (must be distinct, with **different names**)
Play Self: select a single agent to play against itself.
(** Note: a copy will be created, with “@@” appended to the name,
and its player ID will be the constant COPY (-2) **)
Round Robin: select 2+ agents to compete in a round robin tournament
(every combo plays the specified number of games)
Play All: select a single agent to play all of the other agents available in the game

Enter the number of games and click the “+” button to add it to the game queue on the left.

Continue to add games util you are satisfied with your queue.

Click the Green **Start** Button (bottom left).
(** Note: Anything selected in **Additional Settings** and **Pause Conditions** will also be applied **)

Additional Settings

(applied to **both** QuickStart and Start)

Alternate Player Start: eliminate player 1 advantage by alternating which agent is player one each game
Verbose: print the win/loss record to the terminal
Move Timeout: Enter a move timeout in seconds (decimals allowed) **AND** check the option to apply
Auto-Restart: automatically restart the games queue on finish
Pause on Start: start the game in a paused state (to step through moves or get an accurate non-gameboard updating time)
Pause on Illegal Move: pause whenever there has been an illegal move submitted
Layout Option: Select either Player Invoked or Random Override (overrides the human player)

Pause Conditions

(**only** applied to the Start Button)

Select a the two players and the joint conditions to pause under. Then add them to the list on the left.

Human vs AI mode:

Move ant: Select the ant by clicking it, valid possible moves will be highlighted in green. Clicking on a highlighted square will move the ant to that location.

Attack ant: If an ant's move takes it within range of an opponent's ant, then it ***must*** make an attack. This is done by simply clicking the ant to attack. (*Note:* valid attacks options are highlighted in red).

Capture a building: End an ant's turn on an enemy's building. This will reduce the building capture health by 1 and when it reaches 0; either ownership of the building will swap to the capturing player or result in victory if the anthill is captured.

Build ant: Select the anthill when there is no ant currently on it. This will open a menu listing all available ants to build. Ants which the player can afford will be listed in black, ants which the player cannot afford will be listed in red. Selecting an ant from this list will build the ant on the anthill and deduct the food from the player's food supply.

End turn: Simply click the "End" button.

Undo: Click the undo button to undo moves. The undo button can revert the state up through the beginning of the player's turn.

AI vs. AI mode:

AI vs. AI mode is used to test out how two AI players compare in strategy and which one is ultimately more successful.

Actions available

Step: Click the button to see the next action from the AI whose turn it is
Play: This button is only visible when the game is paused. This causes the AIs to play the game at highest possible speed.
Pause: This button is only visible when the game is not paused. This causes the AIs to stop automatically playing.

Hotkeys

Hotkey information is now built into the help menu in the top menu-bar on the screen.

```
<Return>     : end turn
<space>      : step
<p>          : pause/play
```

```
<u>          : undo
```

Developer's Guide
Creating an AI player

Note: Every AI player plays as if it is player 1, i.e. will always appear to be on the side from (0, 0) to (3, 9).

The only class that students will need to edit is the AIPlayer.py class which is located within the "AI" subdirectory of the main aNTiCS folder. They will need to fill out the getPlacement, getMove, getAttack, and registerWin methods in this class. These will be called by Game.py to get the actions from the AI.

getPlacement(currentState):
Parameters:

currentState [GameState] – The current state of the game at the time the Game is requesting a placement from the player.

Expected Return:

If setup phase 1: list of 11 2-tuples of ints -> **[(x1,y1), (x2,y2),…,(x11,y11)]**
If setup phase 2: list of two 2-tuples of ints **-> [(x1,y1), (x2,y2)]**

Description:
The getPlacement method corresponds to the action taken on setup phase 1 and setup phase 2 of the game. In setup phase 1, the AI player will be passed a copy of the state as currentState which contains the board, accessed via currentState.board. The player will then return a list of 11 tuple coordinates (from their side of the board) that represent Locations to place the anthill, tunnel, and 9 grass pieces. In setup phase 2, the player will again be passed the state and needs to return a list of 2 tuple coordinates (on their opponent's side of the board) which represent Locations to place the food sources.

getMove(currentState):
Parameters:

currentState [GameState] – The current state of the game at the time the Game is requesting a move from the player.

Expected Return:

Move(moveType [int], coordList [list of 2-tuples of ints], buildType [int]

Description:
The getMove method corresponds to the play phase of the game and requests from the player a Move object. All types are symbolic constants which can be referred to in Constants.py. The move object has a field for type (moveType) as well as field for relevant coordinate information (coordList). If for instance the player wishes to move an ant, they simply return a Move object where the type field is the MOVE_ANT constant and the coordList contains a listing of valid locations starting with an ant and containing only unoccupied spaces thereafter. A build is similar to a move except the type is set as BUILD, a buildType is given, and a single coordinate is in the list representing the build location. For an end turn, no coordinates are necessary, just set the type as END and return.

getAttack(currentState, attackingAnt, enemyLocations):
Parameters:

currentState [GameState] – The current state of the game at the time the Game is requesting a move from the player.
attackingAnt [Ant] – A clone of the ant currently making the attack.
enemyLocations [list of 2-tuples of ints]
– A list of coordinate locations for valid attacks (i.e. enemies within range)

Expected Return:

(x1, y1) – A coordinate that matches one of the entries of enemyLocations.

Description:
The getAttack method is called on the player whenever an ant completes a move and has a valid attack. It is assumed that an attack will always be made because there is no strategic advantage from withholding an attack. The AIPlayer is passed a copy of

the state which again contains the board and also a clone of the attacking ant. The player is also passed a list of coordinate tuples which represent valid locations for attack. Hint: a random AI can simply return one of these coordinates for a valid attack.

registerWin(hasWon):
Parameters:
hasWon [boolean] – True if the player has won the game, False if the player lost.
Expected Return:
The game expects no return from registerWin().

Description:
The last method, registerWin, is called when the game ends and simply indicates to the AI whether it has won or lost the game. This is to help with learning algorithms to develop more successful strategies.

Class descriptions

Students should not be allowed to edit the classes listed below. However, they will be receiving instances of these classes during gameplay so they should understand how each class works and what each is used for.

AIPlayerUtils.py
This file contains many useful methods that can be used by a developer. These methods include legalCoord, getAntList, getConstrAnt, listAdjacent, etc.These useful methods that give a developer information about a GameState, specific objects, or tuple data sets. Additionally, pathfinding methods are also included in this file which come in varieties such as a greedy algorithm which will perform a less accurate, but faster calculation versus the A-Star algorithm which performs an best optimized path, but can be slower (usually not at such a small scale).

Ant.py:
Ant represents the properties of one of the units in game. This file contains all the stats for ants in a 2-dimensional list called UNIT_STATS. The first index into UNIT_STATS indicates the type of ant while the second represents the desired statistic, e.g. health, range, or movement points.

Building.py:
Buildings represent the player-owned structures that may occupy a tile of the board, anthills and ant tunnels. Building type objects include a capture health field and a player field whereas a general Construction object does not have either.

Constants.py:
This file contains all the named constants used in the game. This is convenient for programmers to include in their AI as it has descriptive names for numeric representations such as list indices and game modes.

Construction.py:
Construction accounts for anything which can occupy a single tile on the board, except Ants. Construction also encompasses the class Building. This file contains all the stats for constructions in a 2-dimensional list called CONSTR_STATS. The first index into CONSTR_STATS indicates the type of contruction while the second the desire statistic, e.g. move cost, capture health, or build cost.

Game.py:
The Game class serves as the central controller for the program and specifically handles tasks relating to the game logic. Its responsibilities include keeping track of the rules, maintaining the official game state, requesting moves from the players on their turns, and making calls to the user interface component which visually renders the GameState.

GameState.py:
The GameState class holds all the information needed to represent the current game and is also used by the game to communicate the information about the game to player. The state contains an instance variable for the board which is a 2-dimensional list of Locations. It also contains an instance variable for the phases of the game, which are menu, setup phase 1,

setup phase 2, and play. The menu phase takes place before the game starts as the player is choosing options. Setup phase 1 begins with a clear board and has the Game requesting coordinate tuples from the Players of where to place the anthill and grass Constructions. Setup phase 2 has the Game requesting coordinate tuples of where to place food Constructions. Once the setup phase is complete, the play phase has the Game requesting Moves from the Players during their turns. The end phase is reached once either Player has reached a victory condition and thus indicates that the game is over. What's expected of the AIPlayer in these phases is explained in more detail in the AIPlayer.py section.

HumanPlayer.py:

The HumanPlayer represents the user when playing in the Human vs. AI game mode. This deals with some elements of the UI and should not be edited by students.

Inventory.py:

Each player's Inventory keeps track of their Ants and the amount of food they have gathered.

Location.py:

Each Location is a single square on the board. When they are all stored in a two dimensional list by the Game_State they make up the game board. Each Location can hold a single Ant and one Construction (a Building, food, or grass).

Move.py:

As the game loop runs, the Game sequentially requests Moves from each Player. Certain fields of the Move are used depending on the type of Move as explained below. The types of moves are: moveAnt, build, and endTurn. A moveAnt Move moves an Ant on the path designated by the Move's coordlist. A build Move builds the unitType of Ant at the first location on the coordlist. An endTurn Move signals that the Player wishes to end the turn and allow the next player to move.

Player.py:

The purpose of the Player class is to handle requests for valid moves from the game logic based on GameState state. This is the class that decides what moves to make during the game. Game.py calls the getMove method whenever it is the player's turn. The AIPlayer.py inherits from this class.

GUIHandler.py:

Manages the three panes and button functionality. This call forms the interface between the game engine and all UI elements. The user should have no reason to interact with this class.

GamePane.py:

Class extends Frame for the game board pane.

StatsPane.py:

Class extends Frame for the stats pane.

SettingsPane.py:

Class extends Frame for the settings pane.

RedoneWidgets.py:

Additional widgets, expanded from tkinter options. Custom button that is colored on all OS platforms.

InfoScraper.py:

Scrapes files for hotkey data and unit stats and any other game rule information.

Glossary

List – a built-in data structure for Python. Lists may be accessed like an array via listVar[0] or using the library methods. Lists may be passed freely and assigned to variables. Unlike tuples, the contents of lists may be modified.

self – by convention, this is included as the first parameter of every function when using Python as an object-oriented language. Unlike languages like Java, the methods and instance variables of the class need to be accessed through self as in self.varName or self.methName(param) instead of just the variable or method name. When calling methods, no argument is provided for the self parameter so every function effectively has one less parameter than it appears.

E.g. The function defined by def someFunction(self,x) would be called by someFunction(2)

Tuple – a built-in data structure for Python. A programmer may simply parenthesize expressions to create tuples. Elements in a tuple do not need to contain the same type, e.g. (1, b, True, None). Tuples may be passed freely and may assigned to variables. Tuples are immutable and cannot be modified once created.

Tuple coordinates – a coordinate represented by a 2-tuple of integers, e.g. (0, 0), (2, 5), (1, 4), etc. These are used frequently by the Game and AIPlayer to represent locations on the board.