CS 228 Fall 2019
Homework 4 (Part 1) - Galaxy Commander

(There is NO LATE DUE DATE) - LATE HWs WILL NOT BE ACCEPTED)

Your task for this homework will be to write a software agent to play an empire building, turn-based simulation game. We will first describe how the game is played and then your responsibilities as an Agent-implementer.

You are required to submit an agent for a grade on homework 5. Your agent must at least be able to capture all the systems in the galaxy when played against an opponent that does nothing. It must also capture more than half of the systems when played against an opponent that does trivial but undisclosed actions (i.e., your agent must be able to win against a reasonably bad agent).

You may also opt to submit your Agent for an all-class tournament to take place on the last day of the semester. If your agent does well, you might win a fabulous prize! Although developing an agent for the tournament will help think about the assignment carefully, it is not required. We will simply use all agent drafts submitted by the tournament deadline with the Agent method inTournament() returning true - you will still be able to revise your submission after this date for grading.

Contents

Basic Game Rules

The game board consists of a galaxy with some number of solar systems and 2 agents. Each agent is initially located at a random solar system. The solar systems can only be reached via one-way wormholes from other systems.

The goal of each agent is to control the most systems when time expires. A standard game is 300 actions — any action an agent takes takes one "agent action".

Intial Setup: Each agent is assigned to an arbitrary system (see below for system characteristics) and given equal energy.

Actions: The agent must decide orders for its next three agent actions (called a "command turn").
Each command turn, the agent receives the results of a probe: a summary of systems out to a user-specified scanning range from where the agent concludes its third action. Distance for scans ("range") is defined, for purposes of scanning, as the number of wormholes you must pass through to get from one system to another (at minimum). Note that this is different from the "cost" to get to that system, which is the sum of the costs of the wormholes you take.

This information includes the following characteristics of each system: whether the opponent agent is present in the system; which agent controls the system; current energy generator count; maximum energy generator count; energy stored; cost to capture (given in energy/turns); what wormholes exist in the system and their destinations; the energy cost to use each wormhole. This is returned in the form of a "state" object from which you can query this information. Note that this information will change from action to action but always reflects the values at the start of the last command turn.

The agent then decides on three agent actions chosen from the following:

1. Move(destination) - Move the agent to the provided destination, expending the requisite energy. If the agent does not have the energy to move, it loses this action (but loses no energy).

2. Refuel() - Transfer all the energy in the system to the agent. This may only be done if the agent owns the system (otherwise, the action is wasted).

3. Fortify(energy) - Increases the cost to capture the system by other players by half (rounded down) of the amount of energy used to fortify the system (overhead costs are a killer). This may only be done if the agent owns the system (otherwise, the action is wasted but energy is not). For example, Fortify(4) increases the cost to capture the system by 2.

4. SetScan(energy) - Sets the amount of energy spent by the probe before each command turn. Before the next command turn, this agent will receive data from probes for all systems within (energy+1) wormholes from where the agent finishes its third action. Initially, all agents are set to expend no energy on probes (i.e. see only the current system and adjacent). Once the scan energy is set, it does not need to be reset each command turn — that is, once an agent does SetScan(3), it will expend 3 energy each turn and get a scan of the current system and all systems up to 4 wormholes away. If, at some point, the agent doesn't have enough energy for its requested level of scan, it will automatically reset to not using any energy (equivalent to calling SetScan(0)).

5. Capture(energy) and ContinueCapture() - Capture the current system by expending the energy cost or the equivalent number of actions to capture, or a combination of the two, making the agent the proud owner of that system. Each system has an associated cost to capture and the agent may capture the system by either expending energy or actions. The agent must expend at least one action capturing a system. Any energy expended beyond the amount needed is lost.
   When a capture begins, the agent spends the energy but, if they did not spend enough energy to immediately capture the system, will need to wait further actions. The agent has the option to continue capturing (through the ContinueCapture() action) or change actions. If an agent stops capturing to take another action, it must restart paying the cost (any loses any initial energy spent). For example, an agent doing Capture(5) in a system with 6 cost will capture it immediately (remember, you always spend one turn capturing). Capture(2) will require the agent to wait 3 additional actions (via ContinueCapture()) or give up on the capture. Capture(8) will just waste 3 extra energy units (but capture the system in one action).

6. Shoot(String[],energy) - Flood a specified system with a low-band ionic pulse from an ionic torpedo. The system's storage and all agents in the system named in the last position of the input array lose energy according

to the following formula: $loss = \lfloor 3 * (energy - 0.25 * \sum_{w \in W} w) \rfloor$ where W is the set of wormhole costs in the given path from the agent's current position to the target system, and the $\sum$ term means "the sum of all the travel costs across the wormholes in W". This means the ionic torpedo will traverse the wormholes incurring a travel cost that is $\frac{1}{4}$ that of a ship, which is deducted from the energy given to the torpedo. The damage inflicted is 3 times that of the energy remaining when the torpedo reaches its destination. An agent or system will be reset to 0 energy if they lose more energy than they have.

The torpedo travels along the path specified by the array of system names input (implicitly starting at the agent's current system). If the given path is invalid, the torpedo will randomly choose wormholes, starting at the last system in the correct portion of the path, and attempt to travel the same number of wormholes desired. If the torpedo does not have sufficient energy to travel through a wormhole, it will detonate at the last system it can reach, regardless of whether it is traveling the requested path or a random path. If the torpedo detonates in the system where the shooting agent is, it may be drained as well! There is no way to defend against this attack but it is executed last during each action (it does take some time for the torpedo to travel).

For example, if an agent (in system "A") shoots at another agent in system "D" with 9 energy but must pass through systems "B" and "C" to get there, with the three wormholes having travel costs of 3, 4 and 5 respectively, the call would be with an array containing "B","C", and "D" and the ionic pulse would drain $\lfloor 3 * (9 - 0.25 * (3 + 4 + 5)) \rfloor = 18$ energy units from any agent in "D" as well as that system's store. However, if, for example, there was no wormhole from "B" to "C", the torpedo would instead travel from "A" to "B", then randomly choose 2 more wormholes to traverse (given that it has enough energy to do so) and detonate on the resulting system, which could be the shooting agent's system!

7. **NoAction()** Perform no action. Note that this is distinct from the **ContinueCapture()** action - if you do **NoAction()** while attempting to capture, your capture is cancelled!

Other Game Notes

Before each action, every agent collects solar radiation, adding 1 energy unit to their energy store.

Every system is reachable from every other system, either directly through a wormhole, or indirectly through other systems.

There may be wormholes that start and end in the same system. There will never be more than one wormhole an agent can take from one system to another.

The cost to use each wormhole will change slightly from action to action due to random galactic fluctuations. This will be an increase or decrease of up to one energy unit cost before each agent action (ie -1,0, or 1 added to the cost). A wormhole will cost at least 1 unit of energy to pass through (the only way a wormhole can seem to cost 0 is if you are attempting to use a wormhole that you have not scanned).

Each system gains 1 energy generator count before each action. If a system is captured, it loses half its generators (rounded up) after the capture action (conquering the planet is always a messy business).

Each system adds 1 energy unit to its store before each action round (immediately after generator increase). It can only store as many energy units as it has energy generator units. If current generator count is reduced while the energy store is not, will not increase until it refuels an agent or the generator count climbs enough.

Initial cost to capture a system will be set by an unknown function depending only on the number of generators. That is, systems with more generators will tend to require more energy/actions to capture but there will be some fluctuation. Once a system is captured, its cost to capture is reset according to the same unknown function. The cost will otherwise not change unless a planet is captured or is reinforced via fortifications.

Both agents may coexist in the same system and attempt actions there. If both are attempting to capture the system, the first player to finish the capture gains control of the system and the other(s) efforts to capture is lost (though they may restart their attempt). If both agents finish at the same time, the winner is resolved arbitrarily (uniformly at random) and the side effects of capturing the system occur twice.

Agents may hold as much energy as you can find (up to Integer.MAX_VALUE).

The exact order of operations during an agent action round is: generator count increase, energy increase, wormhole cost change, all **Move**s, all **Refuel**s, all **Fortify**s, all **SetScan**s, all **Capture/ContinueCapture**s, and (last) all **Shoot**s (shots released by all agents then resolved last in the case of both agents firing).

Example Game

What follows is a simple two agent game given in a simple, text-based format. Here is the initial agent setup:

```
Systems:
 A: Gray,5,10,0,5,{B,C},{1,2}
 B: Gray,2,7,0,10,{C},{3}
 C: Gray,10,20,0,20,{B,E},{1,1}
 D: Gray,3,6,0,8,{A,B},{2,2}
 E: Gray,5,10,0,5,{C,D},{2,1}
Agents:
 Blue: DefaultAgent,A,20
 Red: DefaultAgent,E,20
```

We have 5 systems (A-E) that are specified with: controlling agent (initially "gray" to indicate no agent), current generator count, max generator count, energy stores, cost of capture, the set of wormhole passages that exist, and the cost to use each wormhole. For example, system A is owned by nobody, has generator count 5, max generator count 10, no stored energy, and is connected via wormhole to B and C (with costs 1 and 2, respectively).

There are also two agents whose decisions are made by a class who returns "DefaultAgent" as their names. Displayed also are the color that the agent has (used to tell agents apart), the agent's current system, and it's current stored energy.

Note that, although the full game state is presented, the agents both initially begin scanning with range of 1. So Blue only sees systems A,B, and C while Red sees E,C, and D. What follows is the game state after the Blue player spends 3 (agent) turns and 2 energy capturing A and the Red player spends one turn and 4 energy capturing E, one turn moving to D (costing one energy according to the last scan), and one turn and seven energy capturing D (note that the output of GalaxyCommander on text mode is more verbose than this, giving state after every action):

```
Actions:
 Blue: Capture(2),ContinueCapture(),ContinueCapture()
 Red: Capture(4),Move(D),Capture(7)
Systems:
 A: Blue,4,10,4,6,{B,C},{1,1}
 B: Gray,5,7,3,10,{C},{2}
 C: Gray,13,20,3,20,{B,E},{4,2}
 D: Red,3,6,3,9,{A,B},{3,1}
 E: Red,3,10,3,3,{C,D},{5,2}
Agents:
 Blue: DefaultAgent,A,18
 Red: DefaultAgent,D,7
```

Note that all the above changes are predictable by the agents except for the cost after someone has just captured it (which is set by our unknown function) and the new wormhole costs (which may have changed by up to 3 energy units from last round). Thus, the Red agent lost one more energy making its move to D than specified by its last scan. The next example command turn: Say Blue refuels, moves to C, and spends 20 energy and one turn to capture C. Red fortifies D with 4 energy, refuels, and sets it's scan range to 2 (expend 1 energy/command turn).

```
Actions:
 Blue: Refuel(),Move(C),Capture(20)
 Red: Fortify(4),Refuel(),SetScan(1)
```

```
Systems:
 A: Blue,8,10,2,6,{B,C},{1,2}
 B: Gray,6,7,5,10,{C},{3}
 C: Blue,14,20,5,20,{B,E},{5,3}
 D: Red,6,6,1,11,{A,B},{2,2}
 E: Red,7,10,5,3,{C,D},{4,1}
Agents:
 Blue: DefaultAgent,C,0
 Red: DefaultAgent,D,7
```

Now, since Red has changed its scanning cost, it is charged 1 energy right before making commands to get a range two scan (which enables it to see Blue in system C as well as get information for all systems but E, which is out of range). Blue continues to just see C,B, and E.

Let's say Red shoots at Blue in system C during the first of its agent turns in the next command turn and Blue stays there refueling. It would send the array $[\text{"}B\text{"}, \text{"}C\text{"}]$ and 7 energy. Blue and the system C would lose $\lfloor 3 * (7 - 0.25 * (2 + 3)) \rfloor = 17$ units of energy. Blue would have to start its next turn with no energy and system C would also have no energy (until both gain one at the start of the next turn).

Technical Notes

You will be implementing the cs228hw4.game.Agent interface. Please find a description of the classes for package cs228hw4.game in "Part1Docs.zip". Understanding the enclosed interfaces Agent, GalaxyState, and SystemState (in "Part1Interfaces.zip") will be essential to writing this homework. In order to have access to these interfaces, you must change your Java Build Path (under the properties of a project) to include the external jar GalaxyCommander.jar

For your convenience, the supplied jar file will take command-line parameters to run a text-based user interface. You can find out these parameters by running "*java −jar GalaxyCommander.jar help*" or including the jar file in your classpath (in Eclipse) and running just "*java edu.iastate.cs228.game.gui.GalaxyCommander help*". We recommend you use this interface to save time while testing.

The following conditions should have no effect on most Agent implementations but are included to ensure some fairness in the tournament:

1. No threading, reflection, networking, JNI, or any other technologies that require privileged execution can be used in your Agent implementation unless a good reason is given. This is to eliminate potential for cheating. Disqualification from the tournament is one possible punishment for breaking these rules.

2. Each agent has a maximum of 5 seconds to decide their command turn (the two getCommandTurnX methods). These are real seconds, not CPU seconds. If time expires, the agent does nothing for 3 turns. 5 seconds should be measured on a lab computer and will be interpreted loosely for grading and strictly for the tournament.

3. If your agent throws any unchecked exception (OutOfMemoryError, StackOverflowError, etc.) during a method call, 3 turns are lost and your agent will not be restarted (i.e. if you throw an exception halfway through updating some internal state, you may be hosed).

Other than this, go wild and do whatever you would like. New classes, inner classes, methods, and variables are all fair game. The only thing to consider is that your code will be read and evaluated by the course staff so be sure to use good style and commenting.

No JUnit tests are required. However, they are encouraged and you may feel free to share them on Canvas. As with all homeworks, do not share other, non-test code with other students.

Place all classes you create for this part of the assignment in the package **cs228hw4.game**. Do not create other packages for this part. When you are done, turn in the source files (java files, not class files) in a zipped file (see Part 2).
Don't forget to do part 2 of the assignment as well!