THE IEEE SOUTHEASTCON 2019

SOFTWARE COMPETITION

Friday April 12th

Scrimmage 4:00pm - 7:00pm

Saturday April 13th

Competition 8:00am - 4:00pm

IEEE SOUTHEASTCON 2019
HUNTSVILLE, AL

GOLDEN ANNIVERSARY
POWERED BY THE ROCKET CITY

200

Battlehack: Voyage Official Game Specs

Human civilization on Earth has reached its termination. Fortunately, decades of effort by astronauts, scientists, and engineers seem to have been wildly fruitful, as the explorers they sent into outer space have thrived in their respective landing locations. Inhabitants of the Red Planet — the Zulu Marauders — have regressed to a medieval lifestyle in outer space, living by a feudal system. Inhabitants of the Blue Planet — the Space Marines — are intent on funneling all resources toward researching the ocean life around them. Although both have made countless attempts to build one society, there is no compromising with the Zulu Marauders and Space Marines. With their irreconcilable differences, both civilizations have decided to completely part ways and build separate lives. However, they both have found galactical **Orbs** to be extremely valuable, and are intent on gathering as many as possible. As a Zulu Marauder or Space Marine, you must send out **Voyagers** to explore your area and amass the material around you — specifically, to collect valuable **Orbs** — to build your society.

Game Format

Battlehack: Voyage is a two-player turn-based game, where **Voyagers** on a tiled grid are each controlled by individual computer programs. The objective of the game is to end up with more **Orbs** than the opponent after `256` rounds. **Orbs** are collected by the law of gravity in tiled spaces (i.e., **Orbs** flow towards the closest **Voyager**, in Manhattan distance). Each civilization has a single **Planet** that can spawn **Voyagers**. If by `256` rounds both the Zulu Marauders and the Space Marines have amassed the same number of **Orbs**, the tie is broken by the existing number of units and a coin toss, in that order.

Map

Game maps are procedurally generated, and are square 2D grids ranging between `30x30` and `40x40` tiles. Every map is either horizontally or vertically symmetric, and the top left corner has the coordinates `(0,0)`. Each tile in the map is either passable or impassable; passable tiles are light gray, while obstacles (spaceship corpses from previous exploration, large asteroids, other debris not collectable by the small **Voyagers**) are black. All units have full knowledge of the map from the beginning. Both teams start with one **Planet**, located at symmetric positions on the map.

Orbs

The map is overlaid with a grid of **Orb** counts; each tile contains an integral number of **Orbs** between `0` and `7` (inclusive). Impassable tiles have `0` **Orbs**. The **Orbs** count is constant throughout the entire game, and is known by all units from the start.

At the end of each round, each **Voyager** will use their own gravitational field to pick up all **Orbs** that are closer to them than to all other **Voyagers** (in Manhattan distance).

That is, the total number of **Orbs** collected in a round for one civilization will be the sum of the **Orbs** values of all tiles that are closer to any of the civilization's **Voyagers** than to all of the opposing teams **Voyagers**. **Planets** have too small mass to affect the gravitational field, and thus cannot collect **Orbs**. Neither civilization collects the **Orbs** of tiles that are equally close to their closest **Voyagers** of each civilization. Note that the **Orbs**-assignment process is equivalent to dividing the grid into regions based on the Manhattan-distance Voronoi diagram.

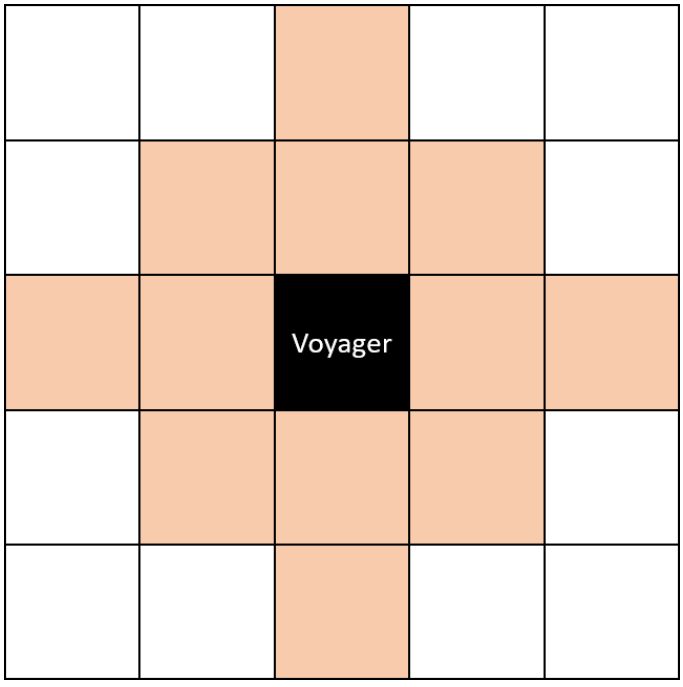
Units

Each unit is initialized with a `100`ms chess clock, and receives `20`ms of additional computation each round. Each turn is additionally capped at `200`ms, after which the code will be stopped. If a robot exceeds its chess clock, it cannot move until it has a positive amount of time in its clock.

Voyagers can move `1` step in the 4 cardinal directions (North, South, East, West). However, **Voyagers** cannot move to a tile that is directly adjacent to another **Voyager** of the same team (diagonals inclusive).

Planets are Voyager factories; they can produce more **Voyagers** to send out into space, but doing so will cost the civilization `65536` **Orbs** per **Voyager**.

Each unit has a unique `4096`-bit integer ID and a vision radius that allows them to see any other unit within a *squared* radius of `4`. This means that a unit at position `(r,c)` can see another unit at position `(r',c')` if and only if `(r-r')^2 + (c-c')^2 <= 4`. Below is a picture showing the visible regions (note that the same is true for **Planets** as well as for **Voyagers**).



Communication

In any given turn, a **Voyager** or **Planet** can broadcast a `16`-bit message to all other units (including the opponent's units) on the map. Until the broadcasting unit's next turn, all units will see the signal broadcasted by it. If a unit does not signal in a given turn, its broadcasted message will be reset to `0`. Units can radio broadcast simultaneously with all other actions.

Turn Queue

Battlehack: Voyage games consist of exactly `256` rounds, and each round consists of a turn for every unit on the board. This is achieved by cycling each round through a queue that consists of all units on the map. The queue is initialized with each team's **Planets** in alternating red, blue order. Then, whenever a **Planet** produces a new **Voyager**, it is added to the end of the turn queue as soon as the **Planet's** turn ends. To rephrase, **Voyagers** built in a round will get a turn in the same round. A round consists of a full pass through the turn queue.

JavaScript Bot Reference

Below is an example of a simple bot. If it is a Voyager, it moves randomly, and if it is a Planet, it tries to build as soon as it can.

```
import {BCAbstractRobot, SPECS} from 'battlecode';

class MyRobot extends BCAbstractRobot {
    turn() {

        if (this.me.unit === SPECS.VOYAGER) {
            const choices = [[0,-1], [1, 0], [0, 1], [-1, 0]];
            const choice = choices[Math.floor(Math.random()*choices.length)]
            return this.move(choice[0], choice[1]);
        }

        else if (this.me.unit === SPECS.PLANET) {
            if (this.orbs >= 65536) {
                return this.buildUnit(0, 1)
            }
        }

    }
}

var robot = new MyRobot();
```

The main container of your bot code is the `MyRobot` class, which must be a subclass of `BCAbstractRobot`.

When your bot is spawned, a `MyRobot` object is created in its own global scope (meaning that you can use global variables, but that they will not be shared between bots). For every turn, the `turn()` method of your class is called. This is where the heart of your robot code lives. If you want the robot to perform an action, the `turn()` method should return it.

Note that the same `MyRobot` class is used for all units. Some API methods will only be available for some units, and will throw an error if called by unallowed units.

You can change the name of the `MyRobot` class, as long as you update the `var robot = new MyRobot();` line.

STATE INFORMATION

- `this.me`: The robot object (see below) for this robot.
- `this.map`: The full map. Boolean grid where `true` indicates passable and `false` indicates impassable. Indexed `[r][c]` (row, column), with `[0][0]` being the upper left corner.
- `this.orbs_map`: The Orbs map. Grid with integer values indicating how much orbs are present at each position. Indexed in the same way as `this.map`.
- `this.n`: The size of the map.
- `this.orbs`: The total amount of Orbs that the team possesses.
- `this.robots`: All units that exist (including `this.me`), in random order.

THE ROBOT OBJECT

Let `r` be any robot object (e.g., `r = this.me` or `r = this.robots[1]`).

The following properties are available for all robots:

- `r.id`: The id of the robot, which is an integer between 1 and 4096.
- `r.unit`: The robot's unit type, where 0 stands for Planet and 1 stands for Voyager.
- `r.signal`: The current signal of the robot.

The following properties are available if the robot is visible (that is, if `isVisible(r)` is `true`).

- `r.team`: The team of the robot, where 0 stands for RED and 1 stands for BLUE.
- `r.r`: The position of the robot in the North-South direction (the row that the robot is in).
- `r.c`: The position of the robot in the East-West direction (the column that the robot is in).

In addition, the following properties are available if `r = this.me`.

- `r.turn`: The turn count of the robot (initialized to 0, and incremented just before `turn()`).
- `r.time`: The chess clock's value at the start of the turn, in ms.

ACTIONS

The following is a list of methods that can be returned in `turn()`, to perform an action. Note that the action will only be performed if it is returned.

- `this.move(dr, dc)`: Move `dr` steps in the North-South direction, and `dc` steps in the East-West direction. Only Voyagers can move.
- `this.buildUnit(dr, dc)`: Build a Voyager in the tile that is `dr` steps in the North-South direction and `dc` steps in the East-West direction from `this.me`. Can only build in adjacent, empty and passable tiles. Uses `65536` Orbs. Only Planets can build.

COMMUNICATION

- `this.signal(value)`: Broadcast `value` to all robots. The `value` should be an integer between `0` and `2^16-1` (inclusive). Can be called multiple times in one `turn()`; however, only the most recent signal will be used.

HELPER METHODS

- `this.log(message)`: Print a message to the command line. You cannot use ordinary `console.log` in Battlehack for security reasons.
- `this.getVisibleRobotMap()`: Returns a 2d grid of integers the size of `this.map`. All tiles outside `this.me`'s vision radius will contain `-1`. All tiles within the vision will be `0` if empty, and will be a robot id if it contains a robot.
- `this.getRobot(id)`: Returns a robot object with the given integer `id`. Returns `null` if such a robot does not exist.
- `this.isVisible(id)`: Returns `true` if and only if the robot identified by `id` is within `this.me`'s vision radius (particularly, `this.me` is always visible to itself)