ICPC Tokyo Regional Contest 2010: Java Challenge

National Institute of Informatics

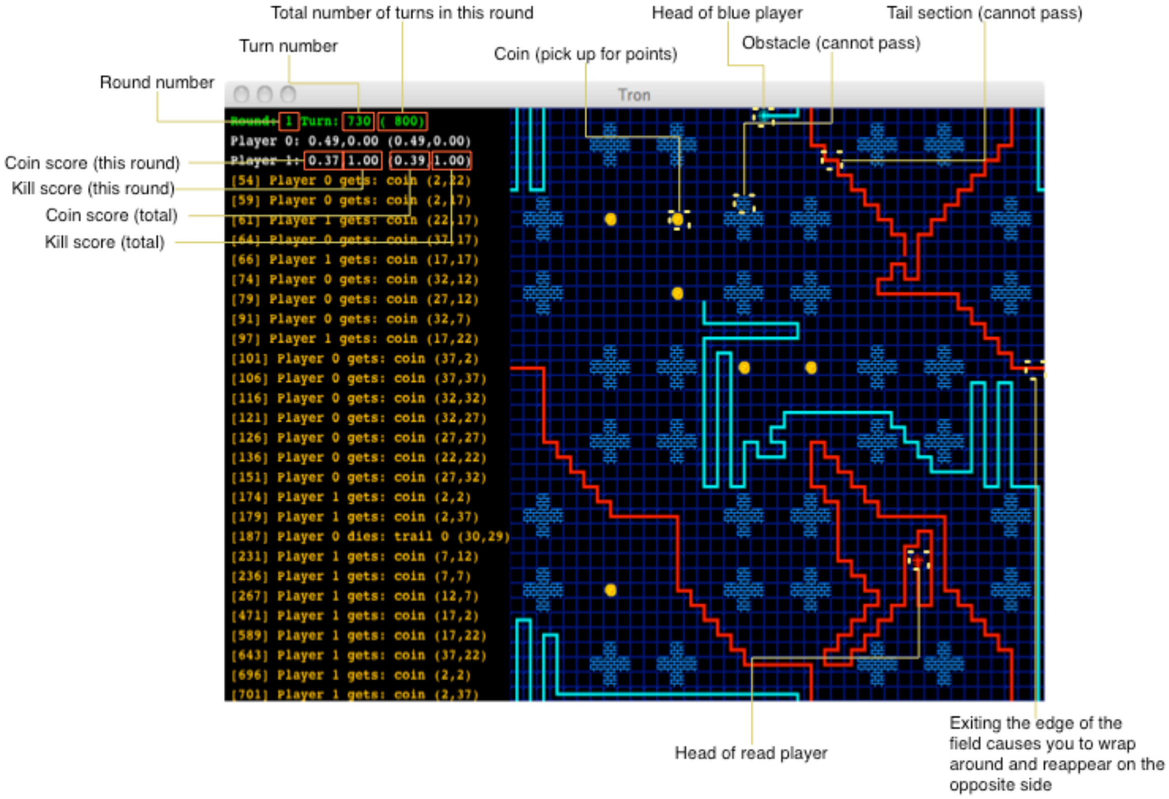
Thank you for joining the ICPC 2010 Java Challenge. As you might already know, the aim of this challenge is to prepare the contestants for the main programming contest, which will take place on the following day. The main league, for ICPC contestants, is called the **contestant league**. In addition, there is also a **special league** with invited guests from the various sponsor companies. Members of the special league have more time to program, since they do not participate in the main ICPC contest. They receive the framework together with this documentation, but members of the contestant league must wait until the 11th of December to receive the framework and start coding.

Introduction

The content of this year's Java Challenge is a game called JTron. This game was developed by a group of students in the Honiden Lab at the National Institute of Informatics in Tokyo. In order to run the game and take a quick look, you can use the following command line (from the directory containing config.xml):

```
java -jar jtron.jar -playerClass CoinPlayer -playerClass
RandomPlayer
```

JTron is inspired by the classic Tron and Pac-man games. The goal in this game is to get as many points as possible within three rounds of the game. To do so, a player should try to survive as long as it can, collect coins, and try to kill its opponent, so the overall gained points of the opponent does not exceed that of the player's.


Total number of turns in this round
Turn number
Round number
Coin (pick up for points)
Head of blue player
Obstacle (cannot pass)
Tail section (cannot pass)
Tron
Coin score (this round)
Kill score (this round)
Coin score (total)
Kill score (total)
Head of read player
Exiting the edge of the field causes you to wrap around and reappear on the opposite side

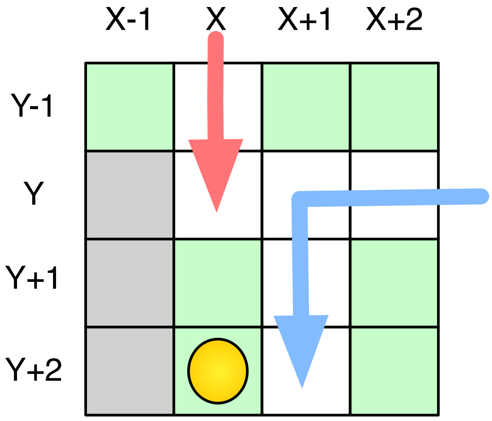
```
Round: 1 Turn: 730 ( 800)
Player 0: 0.49,0.00 (0.49,0.00)
Player 1: 0.37,1.00 (0.39,1.00)
[54] Player 0 gets: coin (2,22)
[59] Player 0 gets: coin (2,17)
[61] Player 1 gets: coin (22,17)
[64] Player 0 gets: coin (37,17)
[66] Player 1 gets: coin (17,17)
[74] Player 0 gets: coin (32,12)
[79] Player 0 gets: coin (27,12)
[91] Player 0 gets: coin (32,7)
[97] Player 1 gets: coin (17,22)
[101] Player 0 gets: coin (37,2)
[106] Player 0 gets: coin (37,37)
[116] Player 0 gets: coin (32,32)
[121] Player 0 gets: coin (32,27)
[126] Player 0 gets: coin (27,27)
[136] Player 0 gets: coin (22,22)
[151] Player 0 gets: coin (27,32)
[174] Player 1 gets: coin (2,2)
[179] Player 1 gets: coin (2,37)
[187] Player 0 dies: trail 0 (30,29)
[231] Player 1 gets: coin (7,12)
[236] Player 1 gets: coin (7,7)
[267] Player 1 gets: coin (12,7)
[471] Player 1 gets: coin (17,2)
[589] Player 1 gets: coin (17,22)
[643] Player 1 gets: coin (37,22)
[696] Player 1 gets: coin (2,2)
[701] Player 1 gets: coin (2,37)
```

As in the Tron game, two players ("agents") consist of colored lines that move across a 2D board ("the play field"). The line ("the trail") grows progressively longer and head of this line is denoted by **Tron**. The trail keeps growing indefinitely, and its length is not limited. A player dies by either running into a wall, into its opponent or into itself. When one player dies, the game still continues for a specified amount of time, and the other player can continue collecting coins and gaining points.

The framework interface is depicted in the image above. The red player is always player 0. The blue player is always player 1. Pressing the keys [**a, s, d, f]** while watching the game will resize the display.

The player can die for the following reasons:

- Entering a square occupied by an enemy trail
- Entering a square occupied by your own trail
- Entering a square occupied by a wall
- Both players enter the same square at the same time (both die in this case)



The above diagram shows which squares are safe for the *red* player to enter. The safe squares are colored green. Entering any other square would result in death (the grey squares are walls). So in this case, the red player can only move *down* during the next turn, and any other move would result in death.

In this game, each **match** consists of three **rounds**, and each round consists of **500 turns**. Each turn lasts for **100 ms**, during which each agent should compute a single move, by using the functions provided by the super-class for getting information on the status of the playfield.

Scoring

Points in each match are calculated in the following way:

- A kill score of 1.00 is given when your opponent dies
- A coin score between 1 and 0, which is given based on the ratio of coins collected in each round to the number of coins available
- The match score is the sum of kill scores and coin scores in the three rounds.

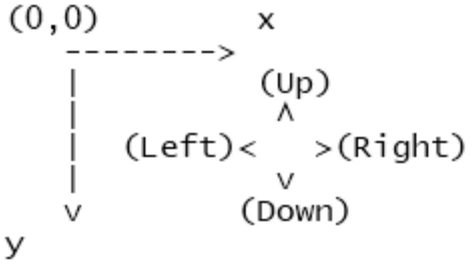
So, a player might die in two rounds of a match, but the sum of its coin score and kill score might be more than its opponents' and ultimately it would win the game.
Note that coins collected by an agent might be regenerated throughout that round, so the coin score would be calculated relative to the actual number of coins generated in that round, not the number visible in the beginning.
The player who has the highest score at the end of three rounds is the winner of the match.

Coding

Player agents should extend the **jp.ac.nii.icpc2010.players.AbstractPlayer** class. Another class called **BasePlayer** is available to help you with some basic utility functions (you can find the source code in the **agents** directory). You should consult the javadoc files provided with this package for more documentation on these and other classes.

For every turn, the agent must indicate the next direction to move. In addition, it can examine every location on the playing field, as well as examine itself and the enemy agent. It can also query for the remaining amount of time and the remaining number of turns.

As you can see for instance in the **RandomPlayer** agent (code included below), agents must override the **getInput()** method, in which they should compute their next move. **getInput()** must return either of FieldDirection.[Up|Down|Left|Right] (see figure below). Agents can use various methods to obtain information needed for this computation. For example, getPlayField(), or simply playField, returns a reference to PlayField. PlayField can be seen as 2-dimentional array which contains objects in the field your tron resides. You can specify a location to query for what object occupies it with **getObjectAt**(int x, int y). This returns either FIELD_FREE, FIELD_COIN, FIELD_WALL or trail ID. Trail ID can be passed to **getTronIdOf**(int trail) to acquire ID of tron that belongs to the trail. There is also a method **getRemainingTime()** available which lets you know how many milliseconds are left of your computation time. If you exceed your computation time limit, which, as mentioned before, is **100 ms**, your agent will repeat its last move by default, which may cause it to die.



```
(0,0)          x
   -------->
   |         (Up)
   |           ^
   |  (Left)<    >(Right)
   |           v
   v        (Down)
y
```

The following is an example of the getInput() method of a sample agent (extending BasePlayer). This agent would select the first move possible in its current position, in the order given below. Note that a reverse move, i.e. 180 degree turn, would kill the agent.

```java
public FieldDirection getInput()
    {
           int curx = getX();
           int cury = getY();

           List<FieldDirection> safeDirs = getSafeDirs(curx, cury);

           if (safeDirs.isEmpty()) {
                //give up
                return prev_dir;
           }

           int dir = (int) (Math.random() * safeDirs.size());

           prev_dir = safeDirs.get(dir);
           return prev_dir;
    }
```

Also, agents must abide by the following design rules:

- Agents must be **single threaded**
- Agents are **not allowed to**: invoke external processes, or do file or network I/O, etc.
- Agent code must be **original and not copied from somewhere else** (we will check, but obviously we cannot check this perfectly, so this is a matter of honour)
- Code must be written in **Java only**, and cannot depend on any libraries or Jar files (**all source code must be provided**)
- Code must not attempt to interfere with the framework, for example by interfering with other threads or with the classloading mechanism, or invoking unofficial methods. **The only acceptable framework interaction is through the official API. Please refer to the javadoc.**
- Agents die if they throw any unexpected exceptions, such as division by 0
- Agents **cannot use an unreasonable amount of memory**. If the JVM runs out of memory, or if the framework starts malfunctioning due to an agent using a very large amount of memory, we will disqualify that agent. **As a rule of thumb, no more than about 40% of the memory limit available to the JVM should be used by the agent.**

For the tournament, we guarantee the following:
Your agent will have exclusive access to a single CPU core running at at least a 2 GHz clock speed.
At least 256 MB of heap memory will be available to your agent.

If you need to, you may create multiple classes for your agent. Every class you create must be **prefixed with your team code.** So for instance, if your team code is T1, then your classes might be called T1Player, T1Helper and T1Extra. **You must submit exactly one class that extends AbstractPlayer or BasePlayer.**

We have provided a template file called YourOwnPlayer.java, which you may use as a starting point if you wish. It is located in the jp/ac/nii/icpc2010/players directory.

Compiling

Note that JTron requires Java 1.6. If this is a problem, please contact us for assistance.

In order for compilation to work, `jtron.jar` must be on your classpath. For instance, if your player source is jp/ac/nii/icpc2010/YourOwnPlayer.java, you might compile like this on a Unix operating system:

```
javac -classpath .:jtron.jar
jp/ac/nii/icpc2010/players/YourOwnPlayer.java
```

Running

In order to run the framework, execute `jtron.jar` using the java interpreter, and pass the name of your class, as well as a competing class, as an input parameter. For instance: `java –classpath .:jtron.jar –playerClass T1Player –playerClass T2Player`

Note that it is possible to play against yourself.

The framework looks for your class in the **jp.ac.nii.icpc2010.players** package by default.

Example opponents

Together with the framework we supply a number of demo classes that you can compete against, and use as a technical reference (the source code is included). Most of them are not very advanced, however, so performing well against the demo classes is no guarantee for doing well in the final tournament.

All of these players are in the jp.ac.nii.icpc2010.players package. The source code can be found in the jp/ac/nii/icpc2010/players directory.

- **HumanPlayer** - you can control the tron yourself using the arrow keys.
- **CoinPlayer** - tries to collect coins using a simple algorithm.
- **Well known Tron strategies:**
  - **WallHuggingPlayer** - Goes close to the walls.
  - **MostOpenDestSelectionPlayer -** Selects the destination with the largest number of open neighboring cells.
  - **RandomPlayer -** Selects a random direction with each move.
  - **OrderedPlayer -** Attempts directions in a specific order.

The four final strategies mentioned here have been adapted from Robert Xiao's Tron strategy guide, which may be a helpful resource (http://csclub.uwaterloo.ca/contest/xiao_strategy.php). However, because JTron also includes features such as obstacles and coins, these strategies may not be sufficient on their own.

Advice

- Try different strategies.
- Make use of all the available CPU time. Use getRemainingTime() to check.
- Taken coins regenerate in the same place after a while.

Tournament Structure

The organization of the tournament is as follows: Each team will first compete in a **group** of four teams in three different **matches**. Two teams from each group proceed to the next **stage** where again groups of four would be formed, until the final stage where the tournament winner is decided. There may be exceptional cases where groups consist of three teams instead of four. In that case, still two teams would advance to the next stage.

On the contest day, the 12th, we will show playbacks from interesting matches, including submissions both from the contestant league and the special league.

Support

Please feel free to contact the Java Challenge team at any time at java-challenge@ml.honiden.nii.ac.jp if you have concerns or questions, or if there is a technical problem with the framework. You can contact us in Japanese or English.