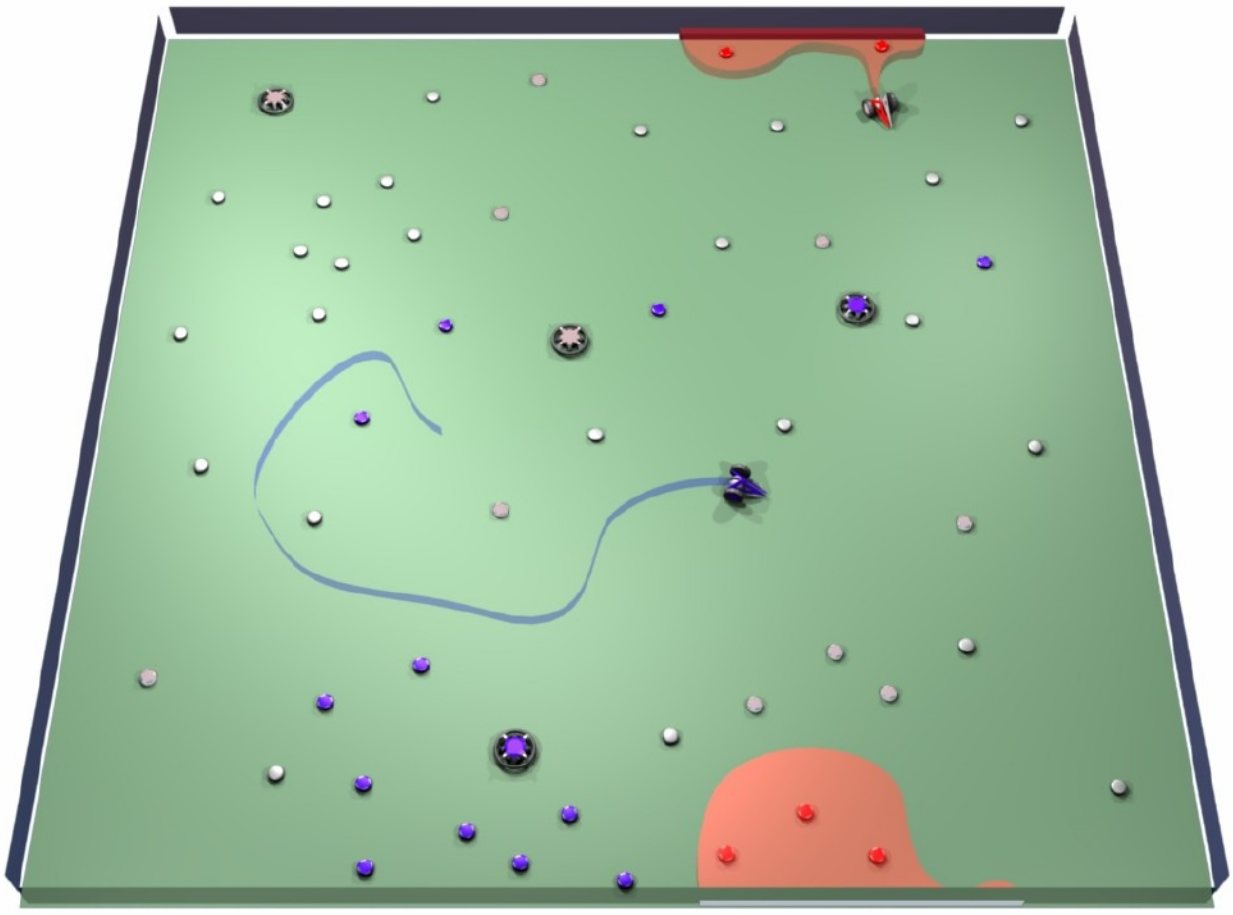
Nfactorial AI Cup

Game Description

April 7-9, 2023

The Capture, is played on a field that looks something like the following figure. A red player and a blue player compete on the 800 800 field, which is populated with 112 pucks of different colors. Players are able to change the color of the pucks, and the goal is to convert as many pucks as possible to the player's own color.



1 Game Pieces

Players participate in the game by controlling three pieces, two bumpers and a sled. The bumpers can be used to move pucks around the field, and the sled is used to change puck colors by drawing a closed loop around them.

1.1 Sled

Each player controls the movement of a sled. The sled moves forward at a constant speed of 15 units per turn. The player controls the sled by adjusting its direction of movement. The sled can turn up to 0.5 radians to the left or right at the start of each turn. Sled motion is not afected by other pieces or the sides of the field. A sled will pass through any other object and will wrap around from one edge of the field to the opposite edge. For example, a sled that drives of the top edge of the field will emerge at the same X coordinate along the bottom edge.

As the sled moves, it leaves behind a polyline *trail*. The trail records the history of where the sled's center has moved over the last several turns. The sled trail does not afect the movement of other pieces, but the sled can alter puck colors by drawing a closed loop around a set of pucks. This is called *capture*. The closed loop formed by a sled can wrap around the edges of the field. In the figure above, for example, the red sled can be seen capturing five pucks, three near the bottom edge of the field and two more near the top edge. Any pucks with centers inside the closed loop are captured. What happens to captured pucks depends on their colors as described below in the "Capture Rules" section.

In the absence of a capture, the sled's trail grows until it reaches 600 units in length. At the end of each turn, older line segments are shortened or removed from the trail until it is no more than than 600 units in length. When capture occurs (even if no pucks are inside the loop), the sled's trail is truncated to remove the closed polyline portion. After this, the trail will continue to grow with the sled's motion until it reaches 600 units or another capture occurs.

1.2 Bumper

Each player controls the behavior of two bumpers. A bumper can move in any direction on the playing field with a speed of up to 24 units per turn. The player controls the bumper by applying a small change in velocity at the start of each turn. The player can change a bumper's velocity by up to 8 units per turn.

Bumpers may collide with pucks, other bumpers and the edges of the field. Collisions are elastic, with bumpers modeled as radius-8 circles with a mass of 8. Bumpers do not experience friction as they move. If the player does not change the direction of a bumper, it will continue to move at the same velocity until it hits something.

1.3 Puck

Pucks are not directly controlled by the player, but they may be moved around with bumpers and captured with sleds. Pucks come in three colors: red, blue and grey. Pucks may experience elastic collision with bumpers, other pucks and the edges of the field. For collision detection, pucks are modeled as radius-5 circles with a mass of 3. Pucks experience a small amount of friction as they move. At the end of each turn, a puck's velocity is reduced by one unit until it reaches zero.

1.4 Capture Rules

When a sled captures a set of pucks, they may change color, depending on the population of the captured set. The result does **not** depend on the color of the sled performing the capture.

- If the set contains only grey pucks, they all remain grey.
- If the set contains only grey and one or more red pucks, they all change to red.

- If the set contains only grey and one or more blue pucks, they all change to blue.
- If the set contains some red, some blue and zero or more grey pucks, they all change to grey.

2 Player/Game Interface

Player implementations are external to the game itself. A player is a separate executable that communicates with the game via standard input and standard output. The player is executed when the game starts up and continues running until the game is finished. At the start of each turn, the game engine sends the player a description of the game state. The player reads this description from standard input, chooses a move and sends it back by writing it to standard output.

2.1 Player Input Format

The game state is sent to the player in a plain-text format. The first line contains a turn number, t. Under normal operation, the value of t will start with zero and increment by one with each game snapshot received by the player. The next line contains a positive integer, p, giving the number of pucks. This is followed by p lines, each describing a single puck. A puck is described by five parameters, floating point X and Y coordinates for the center of the puck, floating point X and Y components of the puck's velocity and an integer indicating the puck's color. The player's color is represented by zero, the opponent's color by one and grey by two.

The list of pucks is followed by a list of bumpers. The bumper list starts with an integer, b, indicating the number of bumpers. This is followed by b lines, each describing a single bumper. A bumper is described by four parameters, floating point X and Y coordinates for the bumper's center and floating point X and Y components of the bumper's velocity. Although the bumper list begins with a count of the number of bumpers, this count will always be four. The list of bumpers is ordered so that the two owned by the player are the first two on the list, and the two owned by the opponent are the second two.

The game state ends with a list of sleds. This list starts with an integer s indicating the number of sleds. This is followed by $2s$ lines, two lines for each sled. A sled description starts with location and orientation information on the first line. The floating point X and Y location of the center of the sled is followed by a floating point rotation value giving the direction the sled is moving. The sled direction is measured counter-clockwise in radians from the positive X axis. This direction reflects the cumulative efect of of the various turns the sled has made, and it will not necessarily fall in the $[\pi, \pi]$ range. For example, if the player constantly makes left turns, the sled direction will only increase.

The second line of a sled description gives a description of the trail behind the sled. This is given as an integer n followed by n floating-point X Y pairs. The trail description starts with the oldest point on the trail and ends with the sled's current location. Wrapping around the edges of the field is indicated by including a point on each edge in the trail. For example, if the sled wrapped around from the left edge of the field to the right edge, the trail would include a point $(0, y)$ followed immediately by a point $(800, y)$.

There are always two sleds. Like the bumpers, the sled descriptions are ordered so that the player's sled is given first and the opponent's sled is given second.

The termination of the game is indicated by a value of -1 for the turn number. Player should read game states and respond with moves until they encounter a turn number of -1.

2.2 Player Output Format

At each turn, the player is to print a desired move to standard output. The move is described by five floating point parameters, the X and Y components of an acceleration vector to be applied to the first bumper, the X and Y components of an acceleration vector to be applied to the second bumper and a desired change in direction for the player's sled. Left turns for the sled are expressed as positive values, and turns to the right expressed as negative values. The five parameters for each move should be printed, space-separated on

a single output line. To ensure that the game is able to read the move immediately after it is produced, the player should flush standard output after the move is printed.

Player moves are applied instantly at the start of the turn. The requested changes in sled direction and bumper velocity are made all at once, and then the next turn is simulated.

If the player requests an acceleration value for a bumper or a turn angle for the sled that is too large, the values are constrained to the legal range. For example, if a player requests that the sled turn -0.6 radians, the sled will turn only -0.5 radians. Acceleration vectors for the bumpers are constrained to 8 in magnitude. If the player provides an acceleration vector that is larger than this, the game will apply an acceleration of 8 in the same direction as the vector provided by the player.

2.3 Real-Time Response Requirement

After a snapshot of the game state is sent, the player generally has 0.1 seconds to respond with a move.
For the first turn of the game, the player has a full second to respond, but subsequent turns give the player only 0.1 seconds. The additional time for the first move reflects the need to give languages like Java an opportunity to demand-load code used by the player. This can cause the first move to take longer than subsequent moves.

If the player fails to respond or if the response is received too late, the game will assign a null move to the player, with all five move parameters set to zero. The game expects to receive a move for each state that is sent to a player, but the game engine does not maintain a queue of game states on behalf of each player. If a player falls behind in parsing game states and responding with a desired move, the engine will discard, rather than queue, subsequent states for the player. A player that is too slow to respond will receive a sampling of the states, and the value of the turn number will indicate that one or more states have been dropped.

At the end of the game, a report is printed to standard output indicating any game states that were discarded without being sent to each player. Likewise, a list is printed reporting any moves that were not received from the player in time.

2.4 Player-Centric Encoding

Communication with the player is encoded so that both players can think of themselves as the red player,
the one that starts out on the left. Internally, the first player given at startup is considered the red player. For the second player, game state is encoded with coordinates for all objects rotated 180 degrees, object lists reversed and colors re-mapped so that both players can believe they are starting on the left edge of the screen. Although the game engine uses zero to represent red, one to represent blue, and two to represent grey, colors in the player's game state description are re-mapped so that zero is the player's own color, one is the opponent's color and two is grey. This is intended to simplify the design of the player somewhat. Developers may wish to hard-code some behaviors with coordinates or color values chosen at compile time.

2.5 Player Debugging

Your player's standard output is used to communicate with the game engine. While developing your player,
you will want to send any debugging output you need to standard error rather than standard output, so that the game engine doesn't think it's part of your move.

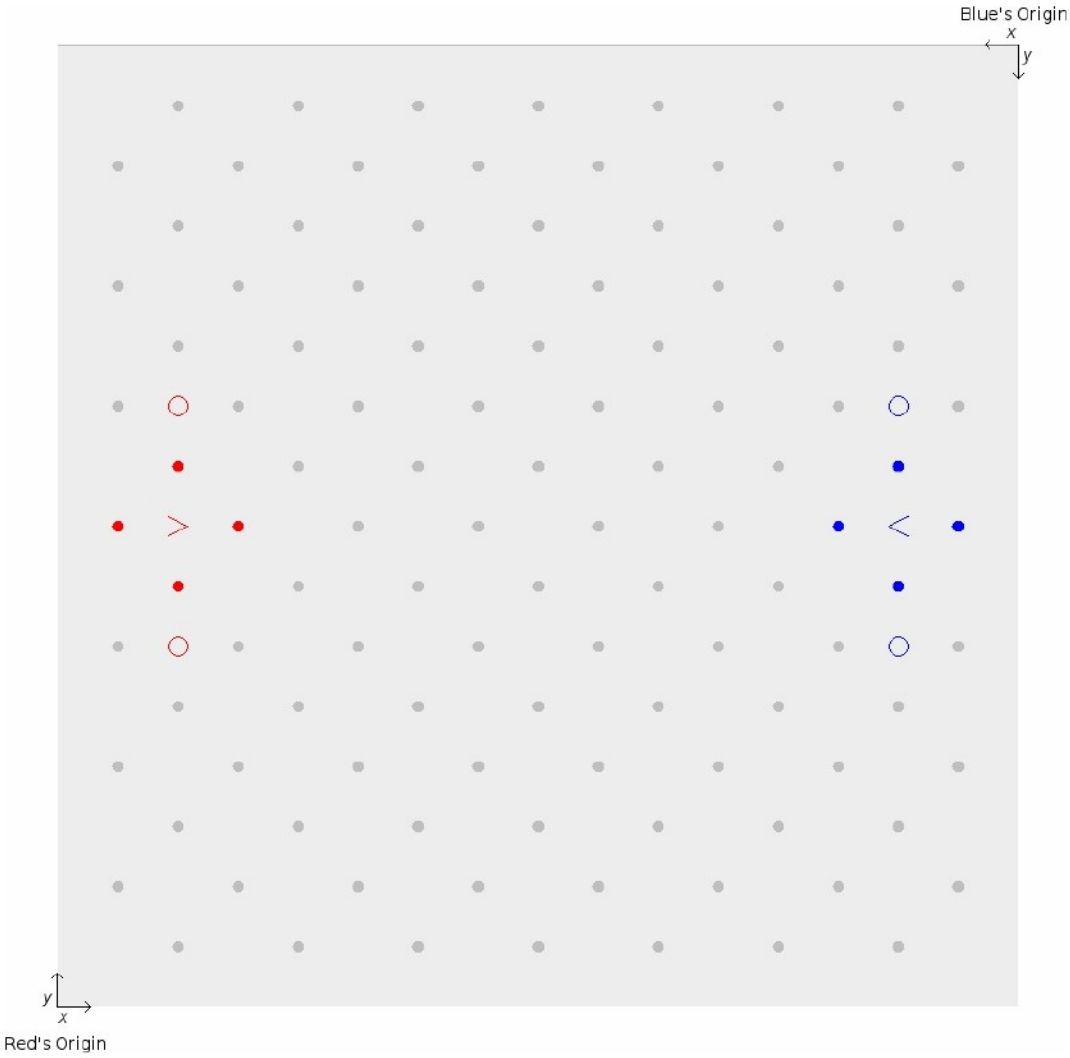
As described later in this document, the game engine can operate synchronously with the player, waiting indefinitely for each move before performing the next simulation step. This lets the developer suspend the real-time response requirement during debugging. The game engine can also be configured to dump game state and player move information for every turn in a game. This can let the developer inspect the sequence of messages exchanged between the game after a game is completed.

3 Gameplay

A match runs for 90 seconds, each player taking 900 turns. At the end, the winner is the player with the most pucks matching his own color. If there is a tie in the number of pucks, puck ownership per turn is summed to break the tie. At the end of each of the 900 turns, we count the number of pucks owned by each player. Ties are broken based on the sum of this count across all turns.

3.1 Initial Field

As illustrated below, the field starts out with one player near the left edge and one near the right edge. The triangles represent the players' sleds, the hollow circles represent the bumpers and the filled circles represent the pucks.



The 112 pucks start out uniformly distributed over the field as illustrated. The puck in the lower left is at location (50, 100), and the rest are distributed in a checkerboard pattern with neighboring pucks displaced 50 units in the X direction and 50 units in the Y direction.

The two sleds start near the left and right edges, with the red sled at location (100, 400) and the blue sled at (700, 400). Both sleds start out pointing toward the middle of the field. The player's two bumpers start

out on either side of the sled, the first red bumper at coordinates (100, 300) and the second at (100, 500). Similarly, the blue bumpers are at locations (700, 300) and (700, 500). Initially, the four pucks around each player's sled are colored with the player's color. The remaining pucks are grey.

3.2 Turn Taking

At the start of each turn, each player receives a copy of the current game state. The game engine waits 0.1 seconds to receive each player's move and then updates the velocity of each player's bumpers and the direction of the sleds. The engine then simulates the next turn of movement. The simulation handles collision and capture on a continuous timescale. These events occur at some time during the 0.1-second turn, not necessarily at the start or the end.

3.3 Frame Interpolation

Players take turns every 0.1 seconds. The 0.1 second delay is intended to give the player an adequate time to select a next move. However, recording animation frames at this rate yields a jumpy playback of recorded games. To help smooth things out, the game engine writes out additional, intermediate states to game trace files. A turn is logged as 3 frames, each reflecting an intermediate state of the turn's simulation. This behavior should not be visible to players participating in the game, but, if ignored, it might cause some confusion about apparent inconsistency between the player's view of the game and the contents of a game trace file.

4 Usage Instructions

The game engine is implemented in Java and supports several command-line options to let the user run games with diferent players and with diferent output behavior. Since players are implemented as separate executables, the developer is free to use any development environment to code the player behavior. In principle, any programming language may be used to implement a player, but the submission site will only accept players written in C++ and Java.

4.1 Making Something Happen

If you just want to see a game running, you can just type the following command. This will start the game with a basic 2D view and without any player code associated with either player. Bumpers will sit still and the sleds will just drive across the field.

```
java -jar capture.jar
```

4.2 Using Built-In Players

If you want to see the players do something a little bit interesting, you can run the game with the following options. This will associate simple, built-in code for each of the two players. The red player will be controlled by a player called RandomPlayer, one that just generates a new, random move every 10 turns. The blue player will be controlled by a copy of LoopyPlayer, one that drives its pieces in circular arcs.

```
java -jar capture.jar -player builtin RandomPlayer -player builtin LoopyPlayer
```

There are three built-in players available, RandomPlayer, LoopyPlayer and DoNothingPlayer. These players don't do much to make the game interesting, but they do provide very simple opponents that should be easy for you to beat.

4.3 Running a C++ Player

Let's say you've implemented a player in C++. You've compiled your player to an executable called `hunter`. You can run this player as the red player using the following command line. You will be playing against a copy of the built-in LoopyPlayer.

```
java -jar capture.jar -player cpp hunter -player builtin LoopyPlayer
```

4.4 Running a Java Player

Let's say you've implemented a player in Java. You've compiled your player to a class called `Gatherer`. You can run this player as the blue player using the following command line. Here, you will be playing against a copy of the built-in RandomPlayer.

```
java -jar capture.jar -player builtin RandomPlayer -player java Gatherer
```

4.5 Running an Arbitrary Player

The above examples of running a player as a C++ executable or as a Java program are really just special cases of the same mechanism. The game engine's `-player pipe` option gives a more general mechanism for running the player executable. This method for starting up a player can let you pass in additional command-line parameters to the player if these are useful during development.

The `pipe` keyword is followed by an integer n. The next n command-line arguments are taken as the command line for the executable that is to serve as the player. For example, the following command line could be used to run the `hunter` C++ program as the blue player and the `Gatherer` Java program as the red player.

```
java -jar capture.jar -player pipe 2 java Gatherer -player pipe 1 hunter
```

4.6 Playing the Game

In single-player mode, the user can direct the actions of one of the game pieces. This can help the programmer to come up with creative strategies by playing one of the pieces and letting their code play the other two. The `-player sled` option lets the user control the sled with the left and right cursor keys. This can be followed by the same options used to specify a built-in or an external player. For example, the following command line will play the built-in LoopyPlayer against the built-in RandomPlayer. However, the user will control the operation of the RandomPlayer's sled.

```
java -jar capture.jar -player builtin LoopyPlayer -player sled builtin RandomPlayer
```

The `-player bumper0` option lets the user control one of the bumpers, while a player implementation controls the other two pieces. For example, the following command line could be used play the built-in RandomPlayer against a C++ player called `hunter`. The C++ implementation will control the sled and one of the bumpers and the the user will control the remaining bumper with the four cursor keys. Although bumpers don't normally experience friction, when a user is controlling one of the bumpers, a small amount of friction is applied after each move in an efort to make the bumper easier to control.

```
java -jar capture.jar -player bumper0 cpp hunter -player builtin RandomPlayer
```

The `-player bumper1` option works just like `-player bumper0`, but lets the user control the other bumper.

4.7 Recording a Game

If you want, you can send a record of game events to a trace file for later viewing. The following command will create a rather large trace file called "trace.txt" containing the sequence of states.

```
java -jar capture.jar -player pipe 2 java Gatherer -player pipe 1 hunter -view
trace trace.txt
```

After you generate a trace file, you can play it back with a trace player. If you've added capture.jar to your CLASSPATH, then the following command will play back this trace using the basic view to show the contents of trace.txt.

```
java icpc.challenge.view.TracePlayer -trace trace.txt
```

4.8 Recording a Turn History

The trace file generated by the `-view trace` option records game event information used by the game's visualization components. It omits some of the information that is available to the players and includes extra information that players don't get to see. The `-view turns` option is intended to capture the sequence of messages exchanged between the game and its two players. Running a game like the following will create a turn history file called "turns.txt" that contains the sequence of states as seen by the red player. In the turn history, each state is followed by the move response that was received from each of the players.

```
java -jar capture.jar -player java Gatherer -player exe      hunter -view turns turns.txt
```

A turn history is intended to help developers debug their players. The file reports game states and moves as seen by the game engine. The re-mapping of colors, game locations and directions normally done when interacting with the blue player is not apparent in this report.

A game can be visualized using its turn file. Since the turn file omits some of the information that's included in a trace, the visualization will not be as smooth and it will not include all efects. However, it can still be useful to give the developer a sense of what was going on at a particular point in the game. If you've added `capture.jar` to your `CLASSPATH`, The following command will play back a game from its turn file.

```
java icpc.challenge.view.TurnPlayer -turns turns.txt
```

5 Example Players

The `c++ example` and `java example` directories contain sample players implemented in C++ and java. The source code for these will demonstrate how a player is to interact with the game, and they may give you some ideas about how your own player could operate. You can compile these independently and then run them against each other in the game using a command like the following:

```
java -jar capture.jar -player exe c++_example/march -player java java_example.Eight
```

Source code for the java is stored in a `java example` subdirectory and is given a matching package name. Teams are reminded that submitted players implemented in Java cannot be organized this way; they must be defined in the default package. The Java example has been organized this way only to simplify distribution.