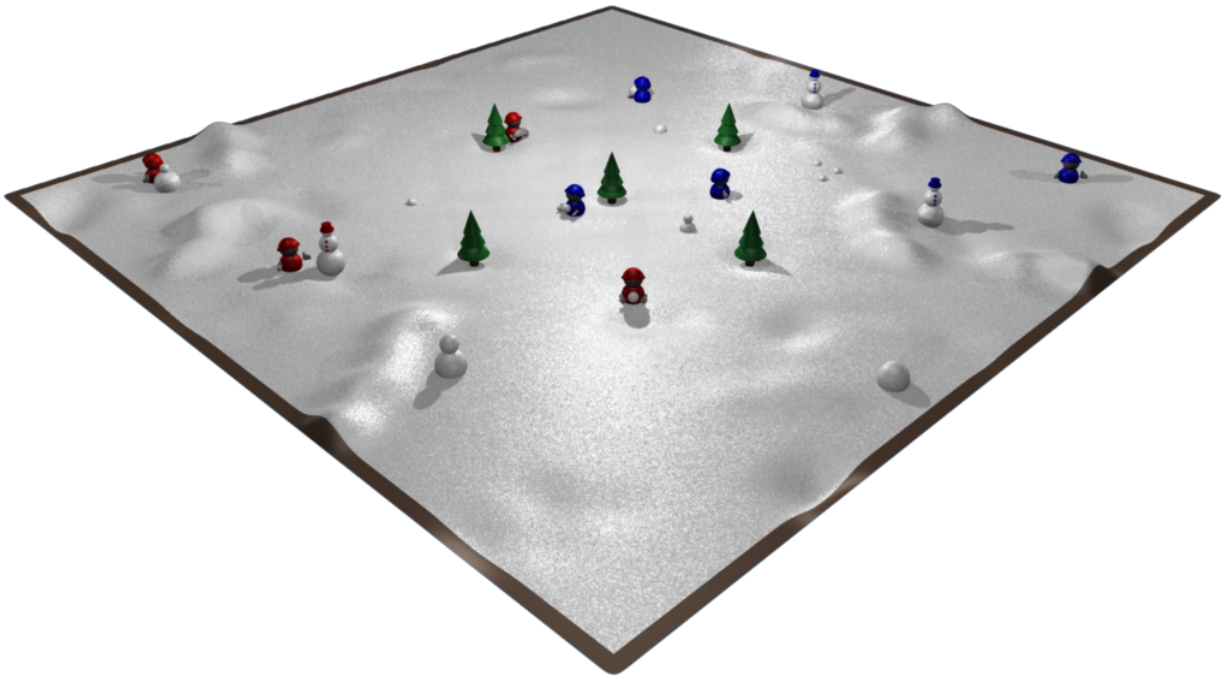
2010 ICPC Challenge

Icy Projectile Challenge Game Description

January 12, 2010

The ICPC Challenge game, Icy Projectile Challenge, is played in an environment that looks something like the following figure. A red player and a blue player compete on a snow-covered 31×31 field. Each player controls a group of children, who can pick up the snow, move it around, make snowballs, hurl them at each other, and stack snowballs to form snowmen. Players earn points by placing snowmen strategically, and by hitting their opponents with snowballs.



1 The Playing Field

Icy Projectile Challenge is played on a field of 31×31 spaces. Spaces are indexed by X and Y, with X going left to right from 0 to 30 and Y going bottom to top from 0 to 30.

A few spaces contain trees. If a space contains a tree, it can't contain any snow, and a tree blocks the movement of children and blocks thrown snowballs. The tree configuration will vary from game to game, but trees will always constitute less than 10 percent of the field spaces, trees will never make non-tree-covered regions of the field unreachable, and a tree will never occupy a child's initial position. To keep things fair, the arrangement of trees will always be symmetric under a 180 degree field rotation. If there is a tree at space (x, y), then there will also be a tree at space $(30 - x, 30 - y)$.

Spaces without trees start out containing loose, powdered snow 3 units high (say, 3 decimeters). As the game is played, snow can be picked up by children, moved around, compressed into snowballs of three different sizes and dropped onto the ground.

1.1 Snow Height and Space Capacity

Each space may contain some amount of snow in the form of powdered snow and, possibly, some snowballs on top. Both powdered snow and snowballs contribute to the height of the snow in a space. A small snowball represents one unit of height, a medium snowball represents two units of height and a large snowball represents three units of height. If, for example, a medium snowball is dropped onto a space already containing powdered snow three units high, the space will now have a height of five. Snowballs are always considered to be on top of any powdered snow in the space, even if the powdered snow was added later.

Each space may contain snow with total height between 0 and 9 units (inclusive). A player is not permitted to drop powdered snow or snowballs onto a space if the additional snow would exceed the 9 unit height limit. A player can't move into or though a space containing snow with a total height of 6 units or more. For example, a child can move through a space containing three units of powdered snow and a medium snowball, but not through a space with three units of powdered snow and a large snowball.

1.2 Snowball Stacking and Space Contents

Snowballs dropped onto the same space can be stacked, one on top of another. If a smaller snowball is dropped onto a space already containing a larger one, the smaller one stacks on top of the larger one. If three snowballs are stacked, they form a snowman owned by the team that dropped the topmost snowball.

If a snowball is dropped onto a snowball of the same size or smaller, it smashes all snowballs of equal or smaller size back into powdered snow in the same space. Likewise, if a child walks through a space containing one or more snowballs, the snowballs are all smashed back into powdered snow. If a snowman is hit in the head with a thrown snowball, its topmost snowball is smashed back into powdered snow in the same space, turning it into a large snowball with a medium one on top.

In addition to some amount of powdered snow, the snowball stacking rules permit a space to contain any of eight different arrangements of snowballs. The following table describes the ten possible contents for a space, in addition to some amount of powdered snow. The right-hand column gives the symbol used to encode each when the game state is communicated to the player[1].

Space Contents	Encoding Symbol
Empty	a
Tree	b
Small snowball	c
Medium snowball	d
Small snowball on medium snowball	e
Large snowball	f
Medium snowball on large snowball	g
Small snowball on large snowball	h
Red snowman	i
Blue snowman	j

[1] A space may also contain a child, but child locations are not encoded as part of the map in game/player communication.

2 Playing The Game

The game proceeds through a series of 180 turns. Each player controls a team of 4 children. At the start of a turn, the player reads a description of state of the field from standard input and then prints a desired action for each child to standard output. One child may directed to pick up snow, while another is making a snowball and a third is moving to a nearby space.

2.1 Child Attributes

Each child may assume either a standing or crouching stance. A child must crouch in order to pick up snow, and it's possible for a snowball to pass over the head of a crouching child. While standing, a child can move faster and can generally throw snowballs further. The height of a child is not affected by the amount of snow in their current space. A child is always six height units tall when crouching and nine height units tall while standing.

Children can also hold snow or snowballs while they move around. The following table describes the nine different things a child can be holding, along with the symbol used to encode each in game/player communication.

Child Holding	Encoding Symbol
Empty Handed	a
One unit of powdered snow	b
Two units of powdered snow	c
Three units of powdered snow	d
One small snowball	e
Two small snowballs	f
Three small snowballs	g
One medium snowball	h
One large snowball	j

2.2 Snowman Domain and Visibility

Players can earn points and strategic advantage by building snowmen on the field. We define the *domain* of player A as the set of board spaces with Euclidean distance less than 8 from a snowman owned by A and closer to a snowman of A than to any snowman owned by the opponent.

At the start of each turn, a player receives a description of the field for any space visible to the player. A player cannot automatically see the entire field. Player A can see any space that's a Euclidean distance of less than 8 from one of the player's children. A player can also see any space that's in the player's domain.

2.3 Getting Hit and Earning Points

Player A earns 10 points when a snowball thrown by a child on the A team hits a child on the opposing team. Player A also earns 1 point for each space that's in the domain of A. As the size of a player's domain increases and decreases, the domain portion of the score will increase and decrease accordingly.

When a child is hit by a snowball (thrown by either team), the child is dazed for the next 4 turns. Dazed children can only perform the idle action, and they appear with a little question mark next to them in the game visualization. If a child is hit with another snowball while dazed, the child will continue to be dazed for four turns starting from the second hit. For example, if a child is hit in turn 5, and then hit again in turn 6, the child will be dazed for turns 6, 7, 8, 9 and 10. Provided there

are no more hits, the child will be free to take an action in turn 11. Dazed children still contribute to field visibility; they just can't take any actions.

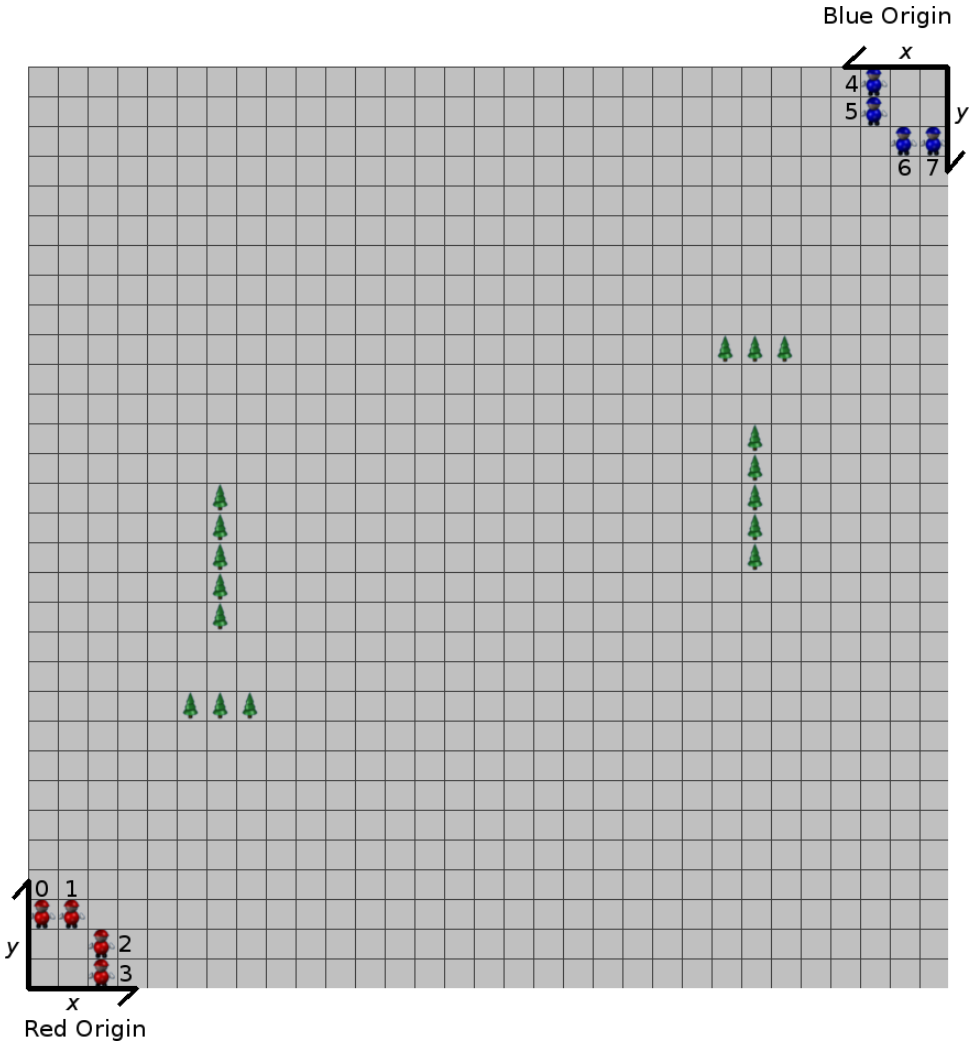
2.4 Winning the Game

At the of the game, the winner is the player with the highest score. If there is a tie in the score, then the player scoring the most points through snowball hits will be the winner. If there is still a tie, the domain size is summed across all turns to break the tie.

3 Directing the Team

Each turn, every child may perform one of ten different actions. All actions require a single turn, although some take effect earlier in the turn than others (see Section 4.1). The player directs children by printing out every child's action, one action per line.

The following figure illustrates the initial placement of children on the field. Children always start out in the positions indicated, and numbers 0 to 3 on the children indicate the ordering of actions for the team. The first action printed out directs child zero, the next action directs child one and so on. Children numbered 4 to 7 are members of the opposing team.



3.1 Idle Action

The idle action directs the child to do nothing. The child stands still for one turn. If a child is requested to perform an action that is not permitted, the idle action is performed instead.

An idle action is encoded as the word "idle" on a line by itself. For example:

```
idle
```

3.2 Crouch Action

If a child is standing, then the crouch action directs a child assume the crouching stance. A crouch action is encoded as the word "crouch" on a line by itself. For example:

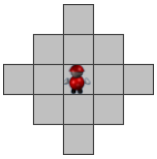
```
crouch
```

3.3 Stand Action

The stand action directs a crouching child to stand up. It's encoded as the word "stand" on a line by itself. For example:

```
stand
```

3.4 Run Action



The run action directs the child to move to a nearby space. A child can run to any space that is no further than a Euclidean distance of two spaces from the child's current location. In order to run, a child must be standing, and the linear path from followed by the child must be unoccupied. A child can move through a space if it doesn't contain a tree, another child or snow of height 6 or greater. Moving to the new space takes a single turn, but it is not instantaneous. For example, if a child moves two spaces to the right, he will briefly occupy the space one to the right along the way. Running to or through a space smashes any snowballs in the space back into powdered snow. For more details on the motion of children and thrown snowballs, see the Linear Path Interpolation section below.

The player encodes the run action as the word "run", followed by the X and Y location of the destination space. For example, to run to a space on the right edge of the field, a player might print out:

```
run 31 10
```

3.5 Crawl Action

The crawl action provides a way of moving while crouching. In a single turn, a child can crawl to the space immediately to the left, right, above or below the current location. The destination space must not contain a tree, another child or snow of height 6 or greater. Crawling to a space smashes any snowballs in the space back into powdered snow.

A crawl action is encoded as the word "crawl", followed by the X and Y location of the destination space. For example, to crawl to a space on the bottom edge of the field, a player might print out:

```
crawl 21 0
```

3.6 Pickup Action

A child can pick up powdered snow or a snowball from any of the eight adjacent spaces, provided the child is crouching and the space isn't occupied by a tree or another child. If the space contains a stack of snowballs, the child picks up the topmost snowball. If the space contains only powdered snow, the child picks up one unit of height in powdered snow, reducing the height of snow in the selected space. Even if the child is already holding something, it may still be possible to pick up something else, provided the result is listed in the table in Section 2.1. For example, a child who is holding one unit of powdered snow can pick up a second unit of snow, and a child who is holding two small snowballs can pick up a third. However, a child holding a large snowball can't pick up anything else without first dropping the snowball.

The pickup action is encoded with the word "pickup", followed by the X and Y location of the destination space. For example, the following line could direct a child at location $(10, 14)$ to pick up snow from an adjacent space.

```
pickup 11 13
```

3.7 Drop Action

A child who is holding something can drop it into any of the eight adjacent spaces, provided the space isn't occupied by a tree or another child, and the additional snow would not increase the height of snow in the space to more than 9. If the player is holding some powdered snow, a medium snowball or a large snowball, then everything is dropped. If the player is holding some number of small snowballs, just one is dropped with each drop action.

If a smaller snowball is dropped into a space already containing a larger one, the smaller snowball is stacked on top of the larger one. If a snowball is dropped onto a space already containing a snowball of the same size or smaller, those snowballs are smashed into powdered snow in the same space.

The drop action is encoded with the word "drop" followed by the X and Y location of the destination space. For example, the following line could direct a child at location $(22, 3)$ to drop snow from an adjacent space.

```
drop 21 4
```

3.8 Crush Action

If the child is holding some amount of powdered snow, the crush action crushes it into a snowball. One unit of snow is crushed into a small snowball, two units are crushed into a medium snowball, and three are crushed into a large snowball. An crush action is encoded as the word "crush" on a line by itself. For example:

```
crush
```

3.9 Throw Action

If a child is holding one or more small snowballs, she can throw one of them. The throw action takes an argument indicating the location of the target space the child is to throw toward. A thrown snowball starts out at the same location as the child throwing it. The snowball starts out at a height of 9 if the child is standing and a height of 6 if the child is crouching. As described in the Linear Path Interpolation section below, a snowball follows a linear path toward the target space. As the snowball travels toward the target, its height also declines linearly, dropping a total of 9 units of height on the way to the target. As it travels, a snowball may hit trees, other children or snow piled up in a space. Typically, a snowball will not make it all the way to the given target space because it will fall into snow on the ground first. At the end of a throw, the snowball is destroyed, and the snow it contained is lost from the field.

The destination space given for a thrown snowball does not need to be on the field. This can be important in considering how the snowball will fall during flight. To hit something, a child may have to throw past it, and this may sometimes require specifying a target space that's off the field.

The target for a throw can be no further than a Euclidean distance of 24 from the child. A throw action is encoded as the word "throw" followed by the X and Y coordinates of the target space. For example, a throw to a space near the center of the field could be encoded as:

```
throw 16 14
```

3.10 Catch Action

If a child is empty handed or holding two or fewer small snowballs, the child is free to catch additional snowballs that are thrown at him. Of course, successfully catching a thrown snowball increases the number of small snowballs held by the child.

Just like the throw, the catch operation requires a target space. The target indicates the location of the child throwing the snowball that is to be caught. A child will catch a snowball if the given target matches the location of the child that threw the snowball and the snowball would have otherwise hit the child.

A catch action is encoded as the word "catch" followed by the X and Y coordinates of the child throwing the snowball. For example, an attempted catch from a player near the upper left corner of the field could be encoded as:

```
catch 30 29
```

4 Simulation Operation

After receiving a player's actions for each child, the game checks the actions to make sure they are legal. If an action can't be performed, if it can't be completed or if it conflicts with an action from another child, the game prints out a brief message and the visualization draws an exclamation mark beside the child.

4.1 Action Ordering and Contention

All actions take only one turn, but they don't all occur at the same time. In a turn, all drop actions are performed first, followed by all pickup actions. Thus, something can be dropped by one child and picked up by another in the same turn. If two children try to drop to the same space in the same turn, both actions are cancelled and the children sit idle. Likewise, if two children try to pick up from the same space in the same turn, both pickup actions are cancelled and the children sit idle.

After the pickup actions, stand, crouch and crush operations are all performed. Finally, run, crawl and throw actions are performed concurrently as a series of one or more single-space steps.

4.2 Linear Path Interpolation

Most actions happen instantly in the game. Movement, however, happens gradually during the execution of a turn. Crawling children, running children and thrown snowballs all follow linear paths in the field, and they all follow the same rule for defining this path and moving along it. They may take several steps to complete their action, and they may pass through intermediate spaces along the way from their starting to their final location.

Consider an entity (either a player or a snowball) that starts at integer position (x_1, y_1) and moves toward position (x_2, y_2) during a single turn. The movement will be implemented as n steps, where $n = \max(|x_2 - x_1|, |y_2 - y_1|)$. The steps will be spaced uniformly in time, with the first step occurring $\frac{1}{n}$ of the way through the turn and subsequent steps occurring at times $\frac{2}{n}, \frac{3}{n}, \ldots \frac{n}{n}$ during the turn. At time $\frac{t}{n}$, the entity moves to integer location $\left(x_1 + \text{round}\left(\frac{t(x_2-x_1)}{n}\right), y_1 + \text{round}\left(\frac{t(y_2-y_1)}{n}\right)\right)$. Thus, the entity will reach

its destination exactly at the end of the turn. Rounding is performed using the following function. While the Java `Math.round()` functions rounds in the positive direction for values halfway between two integers, the function below rounds these values away from zero. This helps to inusre that linear paths are symmetric for the red and the blue player.

```
public static int round( double x ) {
  if ( x < 0 ) {
    return -(int)(Math.round( -x ));
  }

  return (int)(Math.round( x ));
}
```

4.3 Child Movement and Obstruction

Moving children are blocked when they try to move into a space containing a tree, a space containing snow of height at least 6 or a space that contains another child. Until a child encounters an obstacle, she will take as many steps as she can, and then stop moving and wait for a new action at the start of the next turn. If two children need to step into the same space at exactly the same time, then both are blocked and wait where they are until the end of the turn. For example, if a child is two spaces to the right of a tree, the player can still attempt to run two spaces to the left. The child will move toward the tree, taking a first step halfway through the turn. When attempting to take a second step at the end of the turn, the child will be blocked by the tree and will wait next to it until the end of the turn. Obstacles like a tree may be easy to anticipate, but other obstacles may appear as the game unfolds. For example, one child may move to block another or may drop enough snow at the start of a turn to block a child's path.

4.4 Snowball Movement

A thrown snowball follows a linear path to its target space. Since this will usually be a much longer path than that of a moving child, a snowball will typically take more frequent steps during the turn. Also, as the snowball moves, its height drops linearly from the initial height (nine if thrown by a standing child, six if thrown by a crouching child). If a snowball starts out at height h, then, on step $\frac{t}{n}$, the height of a snowball becomes $h - \text{round}\left(\frac{9t}{n}\right)$. Rounding is performed using the procedure described in Section 4.2.

If a snowball's path takes it into a space containing tree or a standing child, the snowball hits the obstacle, shatters and stops there. Hitting other obstacles depends on the height of a snowball. A crouching child is six units high, so a snowball will pass over a crouching child if its height is greater than six while it's in the child's space. Likewise, a snowman may have a height between 6 units (if it's standing right on the ground) and 9 units (if it's standing in three units of powdered snow). If a snowball has the same height as the snowman when it's in the snowman's space, the snowball shatters. This destroys the thrown snowball and reduces the topmost snowball on the snowman back to powdered snow in the space.

If a thrown snowball doesn't hit any other obstacle as it passes through a space, it is permitted to move through the space only if its height is greater than the height of snow in the space. For example, imagine there's a snowman in a space with a total of 7 units of snow. If a thrown snowball has a height of eight as it goes through this space, it passes over the snowman and continues. If the snowball has a height of seven, it hits the snowman in the head. If the snowball has a height of six, it hits the snowman elsewhere and shatters, doing no harm to the snowman.

5 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 (an action for each child) and sends it back by writing it to standard output.

5.1 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 state snapshot received by the player. The next line contains a report of the current game score, first the score for the red player then the score for the blue player.

The score report is followed by a map of the playing field. This is given as 31 rows, each line containing 31 (whitespace-separated) *space descriptions*. The j^{th} space description on line i is a two-character sequence describing the field space with coordinates (i, j). Note that, since row i in the input actually describes spaces with an x coordinate of i, the field description would appear to be rotated by 90 degrees if you looked at it printed out.

If a space is visible to the player, the space description is always a numeric digit followed by a letter. The digit gives the total height of snow in the space, and the letter specifies whether there's a tree or a snowballs in the space as described in the table in Section 1.2. If the space is not visible to the player, then '**' is given as a space description.

The current map is followed by 8 lines, each line describing the state of a child on the field. The first 4 lines describe children on the current player's team and the next 4 describe children on the opponent's team. The ordering of child descriptions matches the order specified in Section 3. For example, child number six in the child description list is always the member of the opposing team that started out at position $(29, 28)$

A child is described by five, space-separated fields. The first two fields give the X and Y coordinates for the child's current location. The next field contains 'C' if the child is crouching and 'S' if the child is standing (both capital). The next field contains a letter from the table in Section 2.1, indicating what the child is holding. The final field is an integer that will be non-zero if the child is dazed. For a dazed child, this value will indicate how many turns must pass before the child can take actions. For example, if this value is one, the child cannot take an action in the current turn, but will be able to in the next turn. If a child is not visible, the child description will be a line containing only a single '*'. Children on the opponent's team may not be visible, but all children on the player's team will always be visible.

5.2 Player Output Format

At each turn, the player is to print a desired action for every child to standard output, one child per line. The ordering of actions is described above in Section 3, and the format for encoding individual actions is described in Sections 3.1 through 3.10.

5.3 Real-Time Response Requirement

After a snapshot of the game state is sent, the player generally has 0.5 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 0.5 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 an idle move to all the player's children. 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.

5.4 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 in the lower left corner of the field. Internally, the first player given at startup is actually considered the red player. For the second player, game state is encoded with coordinates for all objects rotated 180 degrees, child lists reversed and snowman colors re-mapped so that both players can believe they are the red player. This is intended to simplify the design of the player somewhat. Developers may wish to hard-code some behaviors with field locations or other constants chosen at compile time.

5.5 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 game and player after a game is completed.

6 Usage Instructions

The game engine is implemented in Java and supports several command-line options to let the user run games with different players and with different 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.

When you download the game distribution binary, you will receive a zip file containing a small directory structure for the game. You can unpack this file to create an execution environment for the game. The jar file, icypc.jar, contains the executable code for the game, and other subdirectories contain sample map files and example players implemented in C++ and Java.

The icypc.jar file has the main game class (icpc.challenge.main.Game) defined as its entrypoint. You can run matches by simply giving this jar file as the first argument to the Java interpreter. Alternatively, you can run other classes in icypc.jar by putting `icypc.jar` in your `CLASSPATH` and specifying other classes as the entry point.

6.1 Game Visualization

The game distribution binary contains a basic 2D visualization. As described above, this visualization component is intended to help the developer see what's going on in the game, what regions of the field are visible to the player and what actions each child is performing. During the final tournament, we will use a 3D visualization that we hope will provide a more interesting way to watch matches.

6.2 Making Something Happen

If you just want to see a game running, you can unpack the game distribution binary and run the following command from the top-level directory. This will start the game with a basic 2D view of the field and with two example Java players competing against each other.

```
java -jar icypc.jar -player java java_example.Hunter -player java java_example.Planter
```

6.3 Running a C++ Player

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

```
java -jar icypc.jar -player cpp bill -player java java_example.Hunter
```

6.4 Running a Java Player

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

```
java -jar icypc.jar -player java java_example.Planter -player java Lucy
```

Remember that Java players are expected to reside in the default Java package. The example players distributed in the challenge binary are stored in the `java_example` package, just so they don't pollute the top-level directory in the distribution. Players you create should have all their classes in the default package.

6.5 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 C++ program, `bill`, as the blue player and the Java class, `Lucy`, as the red player.

```
java -jar icypc.jar -player pipe 2 java Lucy -player pipe 1 bill
```

6.6 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 game events.

```
java -jar icypc.jar -player java Lucy -player cpp bill \
     -view trace trace.txt
```

After you generate a trace file, you can play it back with a trace player. If you've added icypc.jar to your CLASSPATH, then the following command will play back this trace.

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

6.7 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 icypc.jar -player java Lucy -player cpp bill -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 effects. 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 `icypc.jar` to your `CLASSPATH`, The following command will play back a game from its turn file.

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

6.8 Alternative Maps

By default, the game uses a map with five trees near the center of the field. In the preliminary matches and in the final tournament, tree arrangement may vary from match to match. You can try out your player with alternative tree configurations using the `-map`. This option requires the name of a text file containing an 31×31 image of the initial tree configurations. A period encodes a space without a tree and a capital 'T' encodes a space with a tree. To run a match with an alternative map, you can use a command like:

```
java -jar icypc.jar -player java Lucy -player cpp bill -map maps/map4.txt
```

The `maps` subdirectory in the distribution binary contains a few example maps for you to try. You can build your own using a text editor. Unlike the player input format described in Section 5.1, the map format orients the map like it will appear in the game, with the origin at row 31, column 1 (counting from 1), the X axis going to the right and the Y axis pointing up.

6.9 Reduced View Size

The default view may be too large for users with small screens. To help with this, the game, the trace player and the turn player all accept a `-view simple80` and a `-view simple60` option. These options create a view of the game that is 80 percent or 60 percent of the width and height of the standard view. For example, the following will show a match between the two example Java players using a reduced view size.

```
java -jar icypc.jar -player java java_example.Hunter -player java java_example.Planter \\
     -view simple60
```

6.10 Synchronous Operation

For debugging, players can be started in synchronous mode. This forces the game engine to wait for every move from the player before simulating the next game turn. The following table shows the variants of the `-player` option that can be used to request synchronous operation with a particular player.

Real-Time Response	Synchronous Mode
-player java	-player syncjava
-player cpp	-player synccpp
-player pipe	-player syncpipe

7 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 players independently and then run them against each other in the game using a command like the following:

```
java -jar icypc.jar -player cpp c++_example/camper -player java java_example.Hunter
```

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 just been organized this way only to simplify distribution.