About CodeCraft 2020

General concepts and the rules of the tournament

This competition gives you an opportunity to test your programming skills, by creating an artificial intelligence (strategy) playing a game in a special world (you can learn about details of the CodeCraft 2020 world in later sections). In each game you are to compete against other players' strategies. Your team's goal is to gain more score than your opponents.

The tournament is held in several stages (**Round 1**, **Round 2** and **Finals**) preceded by a qualification in the **Sandbox**. **Sandbox** is a competition that takes place throughout the championship. Each participant has a certain rating value — an indicator of how successful their strategy is involved in games within each stage.

The initial value of the rating in the **Sandbox** is `1200`. At the end of the game this value can both be increased and decreased. At the same time victory over a weak (with a low rating) opponent gives a small increase, also the defeat from a strong opponent slightly decreases your rating. Over time the rating in the **Sandbox** becomes more and more inert, which makes it possible to decrease the impact of random long series of victories or defeats on the participant's place, but at the same time makes it difficult to change their position with a significant improvement in strategy. To cancel such effect the participant can reset the variability of the rating to the initial state when sending a new strategy, including the corresponding option. If the new strategy is adopted, the rating system of the participant will fall dramatically after the next game in the **Sandbox**, however, further participation in games will quickly recover and even become higher if your strategy has really become more effective. It is not recommended to use this option with minor, incremental improvements to your strategy, as well as in cases where a new strategy insufficiently tested and the effect of changes in it is not known reliably.

The initial value of the rating at each main stage of the tournament is `0`. For each game the participant receives a certain number of rating points depending on the occupied place (a system similar to that used in the championship "Formula-1"). If two or more participants share some place, then the total number of rating points for this place and for the following `number_of_such_members - 1` of places is shared equally among these participants. For example, if two participants share the first place, then each of them will receive half of the rating points number for the first and second places. When sharing rounding always takes place in a smaller direction. More detailed information about the stages of the tournament will be provided in the announcements on the project website.

First all participants can participate only in the games that take place in the **Sandbox**. Players can send their strategies to the **Sandbox**, and the last one taken from them is taken by the system for participation in qualifying games. Each player participates in approximately one qualifying game for an hour. The jury reserves the right to change this interval based on the throughput of the testing system, but for the majority of participants it remains constant. There are a number of criteria by which the interval of participation in qualifying games can be increased for a specific player. For every N-th full week that has elapsed since the player sent the last strategy, the interval of participation for this player is increased by N basic test intervals. Only the strategies adopted by the system are taken into account. An additional penalty which is equal to `20%` from the basic testing interval is charged in the **Sandbox** for each strategy "crash" in `10` last games. The player's participation interval in the **Sandbox** can not become bigger than a day.

Games in the **Sandbox** are held according to a set of rules corresponding to the rules of a random stage among the passed ones and the next (current) one. At the same time, the closer the rating value of the two players rating within the **Sandbox**, the more likely that they will be in one game. The **Sandbox** starts before the start of the first stage of the tournament

and ends after some time after the final stage (see the schedule of stages to clarify the details). In addition, the **Sandbox** is frozen during the stages of the tournament. Following the results of the games in the **Sandbox** there is a selection for participation in **Round 1**, which will involve a certain number of participants with the highest rating at the beginning of this stage of the tournament (if the rating is equal, priority is given to the player who previously sent the latest version of their strategy), as well as an additional selection to the next stages of the tournament, including the **Finals**.

Round 1, as all further stages, consists of two parts, between which there will be a short break (with the renewal of the **Sandbox** work), which allows to improve its strategy. The last strategy sent by the player before the beginning of this part is selected for the games in each part. Games are conducted in waves. In each wave, each player participates exactly in one game. The number of waves in each part is determined by the capabilities of the testing system. Highest rated participants will proceed to **Round 2**. Also in **Round 2** there will be an additional selection of participants with the highest rating in the **Sandbox** (at the moment of **Round 2** beginning) among those who did not passed according to the results of **Round 1**.

According to the results of **Round 2** the best strategies will reach the **Finals**. Also in the **Finals** there will be an additional selection of participants with the highest rating in the **Sandbox** (at the beginning of the **Finals**) from those who did not go through the main tournament.

The system of holding the **Finals** has its own peculiarities. The stage is still divided into two parts, but they will no longer consist of waves. In each part of the stage, games will be played between all pairs of **Finals** participants. If the time and capabilities of the testing system permit, the operation will be repeated.

All finalists are ranked according to the non-increase in the rating after the end of the **Finals**. If the ratings are equal, a higher place is taken by that finalist, whose strategy, which was part of the Final, was sent out earlier. Prizes for the Final are distributed based on the occupied place after this ordering.

After the completion of the **Sandbox**, all its participants, except for the **Finals** winners, are ranked according to the non-increase in the rating. If the ratings are equal a higher place is taken by the participant who sent the latest version of their strategy earlier. Prizes for the **Sandbox** are distributed on the basis of occupied place after this ordering.

About testing and strategy limits

Time in the game is discrete and is measured in “ticks”. At the beginning of each tick, the game simulator transmits the world state data to the participants’ strategies, then receives actions from them and updates the state of the world in accordance with these actions and the rules of the game. Then the process is repeated again for next tick with the updated state. There is a maximum duration of the game, but it can also be terminated prematurely if all strategies have “crashed”.

The “crashed” strategy can no longer control player’s behavior. The strategy is considered “crashed” in the following cases:

- The process in which the strategy is started has unexpectedly terminated, or an error has occurred in the protocol of interaction between the strategy and game server.
- The strategy exceeded one of the time constraints assigned to it. There is a time limit for strategy to reply with an action for each tick, as well as a time limit for the whole game.
- The strategy exceeded the memory limit.

Game overview

The game of CodeCraft 2020 is a strategy where you will have to control a number of units, gather resources, build your settlement and attack your enemies.

The game is played on a rectangular grid, divided into tiles. All game entities have square shape and are located at some integer coordinates. When calculating distance, we count the number of tiles that need to be traversed to reach destination, going to a neighboring tile at one time.

Entities' behavior is defined by their properties.

One of the most important properties defines the size of an entity. All entities in the game have square shape, with a side length equal to the defined value.

Some entities can move (these entities are called units). Moving entities always have a size of 1. They can move to a neighboring tile in one tick, if that tile is not occupied by some other entity.

Some entities can attack, and all entities have health and can be destroyed. If entity's health gets below or equal to zero, it is removed from the game. Attacking entities have a limited attacking range, which is a maximum distance to the target for performing the attack. Each tick an entity is attacking, it subtracts a certain amount of health points from the target.

Also, some entities can repair other entities. Only adjacent entities can be repaired. Repairing is restoring a specified amount of health points in on tick. When repairing, target's health can not go above its maximum health, specified in its properties.

Some of the attacking entities can also collect resources from the target. For each health point of damage, a fixed amount of resource (specified in target entity's properties) is added to the attacker's owning player.

Gathered resources can be used to build new entities. Some entity types can do that. New entity's type is limited by capabilities of the builder, listed in its properties. To build a new entity, you have to spend a specified amount of resources. You should also select a location not occupied by other entities, and located not further than build range of the entity performing the building action. Newly built entities have initial health equal either to entity's maximum health, or to a specific value, as specified in builder entity's properties.

When an entity is just built, it is inactive at first, meaning it can not perform any actions. To activate an entity, it has to reach its maximum health first. So, if an entity was built not with full health, it needs to be repaired first.

Also, there is one more restriction to building new entities. Besides resources, there is another parameter called "population". Some entities provide population, while others use it. In order to build a new entity, the sum of population provided by all current player's active entities should be greater than or equal to the sum of population used by all current player's entities, including newly built one.

The last property of an entity is its sight range. If fog of war is enabled, your strategy can only see those entities that are located no further than distance specified from some entity that is controlled by you.

List of entity types

There is a fixed number of entity types in the game, and entities of same type have same properties. Here is a full list of entity types:

- Resource. This is the only entity type that is not controlled by any player. It has a size of 1 and should be attacked by builder units in order to be gathered.
- Builder unit. Its primary purpose is to gather resources and construct buildings.
- Melee unit. Basic damage dealing unit with an attack range of 1.
- Ranged unit. Deals less damage than melee unit, but has bigger attack range.
- Builder/Melee/Ranged bases. These are buildings used to buy new units of corresponding type. Can be built by a builder unit.
- Wall. A small building that can be used to block path for enemy.
- House. A building that provides population.
- Turret. A building that can attack enemies. Since it can not move, its best for defence.

Control interface

Each tick your strategy is going to be asked for actions of your entities. If you do not specify action for an entity, it continues to perform previously set action.

Entity's action consists of attack, build, repair and move actions, which are prioritized in this order. So, if you specify multiple actions, then only the first one that is possible to do this current tick is actually performed.

Attack action can specify a specific entity to attack, or perform an auto attack. When using auto attack, you can also specify a distance where your unit should try to pathfind to try and find an enemy to attack.

Repair action can only specify an entity to repair.

For build action you would need to specify a type of entity you want to build as well as position. Position of an entity is it's corner with minimal coordinates.

For move action target position needs to be specified. A unit will try to pathfind to this position. You can further control pathfinding by specifying whether to find closest position. In case you do not want this, unit will not move if the path is not found. You may also specify whether to try to find such a route which involves breaking through other entities, attacking and destroying them on the way. Such algorithm will only consider to attack entities not controlled by you.

When pathfinding is needed to be performed by the game server, a simple A* algorithm is used, with limited number of nodes to be visited.

Each game tick, first a random order is chosen for entities (the order is different each tick). Then, all attack actions are performed for active entities in this order. Next, build actions are performed in same order (if an entity has performed an attack action this tick, build action will not be performed). Then repair actions and move actions are performed in same way. When performing pathfinding for movement, game's state at the beginning of tick is used. In the end, entities with zero health are removed from the game, and entities with full health become active.

Round specific rules

In **Round 1** you are to learn the rules of the game. To simplify things, there will be no fog of war in this stage. Also, you will be given a base for each unit type in the beginning of the

game, so you can start gathering resources and attacking your opponent right away. You can still experiment with buildings to prepare for next stages.

In **Round 2** you have to learn building. In the beginning you will only have builders. You will need to build bases for other types of units. Also, fog of war will now be enabled so you will have to do some exploration before confronting the enemy. The task is further complicated that after summarizing the **Round 1**, the part of the weak strategies will be eliminated and you will have to confront stronger opponents.

Finals is the most important stage. After the selection, held following the results of the first two stages, the strongest participants will be remained. The games in the finals are going to be 1v1, but otherwise no new rules are going to be introduced.

API description

In language pack for your programming language you can find file named `MyStrategy.<ext>`/`my_strategy.<ext>`. This file contains class `MyStrategy` with `get_action` method, where your strategy's logic should be implemented.

This method will be called each tick.

The method takes following arguments:

- Player view — all the information you have about current game's state,
- Debug interface — this object allows you to do send debug commands to the app and receive debug state from inside your strategy code. Note that this is unavailable when testing your strategy on the server, or using the app in batch mode. This is for local debugging only.

The method should return the action you desire to perform this tick.

For debugging purposes, there is also another method — `debug_update`, that has same parameters, and is called continiously while the app is running (not in batch mode), if the client is waiting for the next tick. There will always be at least one debug update between ticks.

Objects description

In this section, some fields may be absent (denoted as `Option<type>`). The way this is implemented depends on the language used. If possible, a dedicated optional (nullable) type would be used, otherwise other methods may be used (like a nullable pointer type).

Some objects may take one of several forms. The way it is implemented depends on the language. If possible, a dedicated sum (algebraic) data type is used, otherwise other methods may be used (like variants being classes inherited from abstract base class).

`Vec2Float32`

2 dimensional vector.

Fields:

- `x`: `float32` - `x` coordinate of the vector
- `y`: `float32` - `y` coordinate of the vector

`Color`

RGBA Color

Fields:

- `r`: `float32` - Red component
- `g`: `float32` - Green component
- `b`: `float32` - Blue component
- `a`: `float32` - Alpha (opacity) component

`ColoredVertex`

Vertex for debug rendering

Fields:

- `world_pos`: `Option<Vec2Float32>` - Position in world coordinates (if none, screen position (0, 0) is used)
- `screen_offset`: `Vec2Float32` - Additional offset in screen coordinates
- `color`: `Color` - Color to use

`PrimitiveType`

Primitive type for debug rendering

Variants:

- `Lines` - Lines, number of vertices should be divisible by 2
- `Triangles` - Triangles, number of vertices should be divisible by 3

`DebugData`

Debug data can be drawn in the app

One of:

- `Log` - Log some text

  Fields:

  - `text`: `string` - Text to show
- `Primitives` - Draw primitives

  Fields:

  - `vertices`: `[ColoredVertex]` - Vertices
  - `primitive_type`: `PrimitiveType` - Primitive type
- `PlacedText` - Draw text

  Fields:

  - `vertex`: `ColoredVertex` - Vertex to determine text position and color
  - `text`: `string` - Text
  - `alignment`: `float32` - Text alignment (0 means left, 0.5 means center, 1 means right)
  - `size`: `float32` - Font size in pixels

`DebugCommand`

Debug commands that can be sent while debugging with the app

One of:

- `Add` - Add debug data to current tick

  Fields:

  - `data`: `DebugData` - Data to add
- `Clear` - Clear current tick's debug data

  No fields

`Vec2Int32`

2 dimensional vector.

Fields:

- `x`: `int32` - `x` coordinate of the vector
- `y`: `int32` - `y` coordinate of the vector

`MoveAction`

Move action

Fields:

- `target`: `Vec2Int32` - Target position
- `find_closest_position`: `boolean` - Whether to try find closest position, if path to target is not found
- `break_through`: `boolean` - Whether to destroy other entities on the way

`EntityType`

Entity type

Variants:

- `Wall` - Wall, can be used to prevent enemy from moving through
- `House` - House, used to increase population
- `BuilderBase` - Base for recruiting new builder units
- `BuilderUnit` - Builder unit can build buildings
- `MeleeBase` - Base for recruiting new melee units
- `MeleeUnit` - Melee unit
- `RangedBase` - Base for recruiting new ranged units
- `RangedUnit` - Ranged unit
- `Resource` - Resource can be harvested
- `Turret` - Ranged attacking building

`BuildAction`

Build action

Fields:

- `entity_type`: `EntityType` - Type of an entity to build
- `position`: `Vec2Int32` - Desired position of new entity

`AutoAttack`

Auto attack options

Fields:

- `pathfind_range`: `int32` - Maximum distance to pathfind
- `valid_targets`: `[EntityType]` - List of target entity types to try to attack. If empty, all types but resource are considered

`AttackAction`

Attack action

Fields:

- `target`: `Option<int32>` - If specified, target entity's ID
- `auto_attack`: `Option<AutoAttack>` - If specified, configures auto attacking

`RepairAction`

Repair action

Fields:

- `target`: `int32` - Target entity's ID

`EntityAction`

Entity's action

Fields:

- `move_action`: `Option<MoveAction>` - Move action
- `build_action`: `Option<BuildAction>` - Build action
- `attack_action`: `Option<AttackAction>` - Attack action
- `repair_action`: `Option<RepairAction>` - Repair action

`Action`

Player's action

Fields:

- `entity_actions`: `Map<int32 -> EntityAction>` - New actions for entities. If entity does not get new action, if will continue to perform previously set one

`ClientMessage`

Message sent from client

One of:

- `DebugMessage` - Ask app to perform new debug command

  Fields:

  - `command`: `DebugCommand` - Command to perform
- `ActionMessage` - Reply for ServerMessage::GetAction

  Fields:

  - `action`: `Action` - Player's action
- `DebugUpdateDone` - Signifies finish of the debug update

  No fields
- `RequestDebugState` - Request debug state from the app

  No fields

`BuildProperties`

Entity's build properties

Fields:

- `options`: `[EntityType]` - Valid new entity types
- `init_health`: `Option<int32>` - Initial health of new entity. If absent, it will have full health

`AttackProperties`

Entity's attack properties

Fields:

- `attack_range`: `int32` - Maximum attack range
- `damage`: `int32` - Damage dealt in one tick
- `collect_resource`: `boolean` - If true, dealing damage will collect resource from target

`RepairProperties`

Entity's repair properties

Fields:

- `valid_targets`: `[EntityType]` - Valid target entity types
- `power`: `int32` - Health restored in one tick

`EntityProperties`

Entity properties

Fields:

- `size`: `int32` - Size. Entity has a form of a square with side of this length
- `build_score`: `int32` - Score for building this entity
- `destroy_score`: `int32` - Score for destroying this entity
- `can_move`: `boolean` - Whether this entity can move
- `population_provide`: `int32` - Number of population points this entity provides, if active
- `population_use`: `int32` - Number of population points this entity uses
- `max_health`: `int32` - Maximum health points
- `cost`: `int32` - Cost to build this entity type
- `sight_range`: `int32` - If fog of war is enabled, maximum distance at which other entities are considered visible
- `resource_per_health`: `int32` - Amount of resource added to enemy able to collect resource on dealing damage for 1 health point
- `build`: `Option<BuildProperties>` - Build properties, if entity can build
- `attack`: `Option<AttackProperties>` - Attack properties, if entity can attack
- `repair`: `Option<RepairProperties>` - Repair properties, if entity can repair

`Player`

Player (strategy, client)

Fields:

- `id`: `int32` - Player's ID
- `score`: `int32` - Current score
- `resource`: `int32` - Current amount of resource

`Entity`

Game entity

Fields:

- `id`: `int32` - Entity's ID. Unique for each entity
- `player_id`: `Option<int32>` - Entity's owner player ID, if owned by a player
- `entity_type`: `EntityType` - Entity's type
- `position`: `Vec2Int32` - Entity's position (corner with minimal coordinates)
- `health`: `int32` - Current health
- `active`: `boolean` - If entity is active, it can perform actions

`PlayerView`

Information available to the player

Fields:

- `my_id`: `int32` - Your player's ID
- `map_size`: `int32` - Size of the map
- `fog_of_war`: `boolean` - Whether fog of war is enabled

- `entity_properties`: `Map<EntityType -> EntityProperties>` - Entity properties for each entity type
- `max_tick_count`: `int32` - Max tick count for the game
- `max_pathfind_nodes`: `int32` - Max pathfind nodes when performing pathfinding in the game simulator
- `current_tick`: `int32` - Current tick
- `players`: `[Player]` - List of players
- `entities`: `[Entity]` - List of entities

`ServerMessage`

Message sent from server

One of:

- `GetAction` - Get action for next tick

  Fields:

  - `player_view`: `PlayerView` - Player's view
  - `debug_available`: `boolean` - Whether app is running with debug interface available

- `Finish` - Signifies end of the game

  No fields

- `DebugUpdate` - Debug update

  Fields:

  - `player_view`: `PlayerView` - Player's view

`Camera`

Camera used for rendering

Fields:

- `center`: `Vec2Float32` - Center point at which camera is looking
- `rotation`: `float32` - Rotation angle
- `attack`: `float32` - Attack angle
- `distance`: `float32` - Distance to center
- `perspective`: `boolean` - Whether perspective is applied

`DebugState`

Debug state to be received from the app

Fields:

- `window_size`: `Vec2Int32` - Size of the drawing canvas
- `mouse_pos_window`: `Vec2Float32` - Mouse position in window coordinates
- `mouse_pos_world`: `Vec2Float32` - Mouse position in world coordinates
- `pressed_keys`: `[string]` - Currently pressed keys
- `camera`: `Camera` - Current camera used for rendering
- `player_index`: `int32` - Your player's index