Project
Artificial Intelligence for Games

Hubert Obrzut, Szymon Kosakowski

February 19, 2021

1 Introduction

This report describes our project for `Artificial Intelligence for Games` course. We attempted to create a bot for `TORCS` game - car racing simulation and created, implemented and tested a few agents for `Hypersonic` game from **CodinGame** platform. All the bots and agents were implemented in `C++`.

2 TORCS

2.1 Connecting to the game

The game provides a patch for itself which overlaps the game with server to which user can connect via client program. Server sends information about current state of the car user controls using UDP datagrams. User has following sensors at his disposal:

Name	Range (unit)	Description
angle	[-π,+π] (rad)	Angle between the car direction and the direction of the track axis.
curLapTime	[0,+∞) (s)	Time elapsed during current lap.
damage	[0,+∞) (point)	Current damage of the car (the higher is the value the higher is the damage).
distFromStart	[0,+∞) (m)	Distance of the car from the start line along the track line.
distRaced	[0,+∞) (m)	Distance covered by the car from the beginning of the race
focus	[0,200] (m)	Vector of 5 range finder sensors: each sensor returns the distance between the track edge and the car within a range of 200 meters. When `noisy` option is enabled (see Section 7) sensors are affected by i.i.d. normal noises with a standard deviation equal to the 1% of sensors range. The sensors sample, with a resolution of one degree, a five degree space along a specific direction provided by the client (the direction is defined with the *focus* command and must be in the range [-90,+90] degrees w.r.t. the car axis). Focus sensors are not always available: they can be used only once per second of simulated time. When the car is outside of the track (i.e., pos is less than -1 or greater than 1), the focus direction is outside the allowed range ([-90,+90] degrees) or the sensors has been already used once in the last second, the returned values are not reliable (typically -1 is returned).
fuel	[0,+∞) (l)	Current fuel level.
gear	{-1,0,1,$\cdots$ 6}	Current gear: -1 is reverse, 0 is neutral and the gear from 1 to 6.

Figure 1: Description of the available sensors (part I). Ranges are reported with their unit of measure (where defined)

lastLapTime	[0,+∞) (s)	Time to complete the last lap
opponents	[0,200] (m)	Vector of 36 opponent sensors: each sensor covers a span of 10 degrees within a range of 200 meters and returns the distance of the closest opponent in the covered area. When `noisy` option is enabled (see Section 7), sensors are affected by i.i.d. normal noises with a standard deviation equal to the 2% of sensors range. The 36 sensors cover all the space around the car, spanning clockwise from -180 degrees up to +180 degrees with respect to the car axis.
racePos	$\{1,2,\cdots,N\}$	Position in the race with respect to other cars.
rpm	[0,+∞) (rpm)	Number of rotation per minute of the car engine.
speedX	(−∞,+∞) (km/h)	Speed of the car along the longitudinal axis of the car.
speedY	(−∞,+∞) (km/h)	Speed of the car along the transverse axis of the car.
speedZ	(−∞,+∞) (km/h)	Speed of the car along the Z axis of the car.

Figure 2: Description of the available sensors (part II). Ranges are reported with their unit of measure (where defined)

track	[0,200] (m)	Vector of 19 range finder sensors: each sensors returns the distance between the track edge and the car within a range of 200 meters. When `noisy` option is enabled (see Section 7), sensors are affected by i.i.d. normal noises with a standard deviation equal to the 10% of sensors range. By default, the sensors sample the space in front of the car every 10 degrees, spanning clockwise from -90 degrees up to +90 degrees with respect to the car axis. However, the configuration of the range finder sensors (i.e., the angle w.r.t. to the car axis) can be set by the client once during initialization, i.e., before the beginning of each race. When the car is outside of the track (i.e., pos is less than -1 or greater than 1), the returned values are not reliable (typically -1 is returned).
trackPos	(−∞,+∞)	Distance between the car and the track axis. The value is normalized w.r.t to the track width: it is 0 when car is on the axis, -1 when the car is on the right edge of the track and +1 when it is on the left edge of the car. Values greater than 1 or smaller than -1 mean that the car is outside of the track.
wheelSpinVel	[0,+∞] (rad/s)	Vector of 4 sensors representing the rotation speed of wheels.
z	[−∞,+∞] (m)	Distance of the car mass center from the surface of the track along the Z axis.

Figure 3: Description of the available sensors (part III). Ranges are reported with their unit of measure (where defined)

The actions user can make are:

- press gas pedal - value from $[0, 1]$
- press brake pedal - value from $[0, 1]$
- change the gear
- rotate the steering wheel - value from $[-1, 1]$, max rotation is equivalent to 0.366519 rad of wheel rotation.

After understanding the problem, we have decided to try using *Monte Carlo Tree Search* algorithm. With that we decided to stop using the client (written in Python) and connect directly to the game code (in `C++`).

2.2 General approach

As mentioned earlier we used *MCTS* algorithm. We wanted to collect the data of current track on the first lap and then ride on it on next laps with our algorithm. Our idea was to ride slowly at the center of the track on the first lap and collect the coordinates of driving line on every iteration. It quickly turned out that we need to come up with a simulation of the car first to collect the data (the client didn't have access to the position of the car) and second to simulate in *MCTS*.

2.3 Physical attempt

At first we tried physical simulation. `TORCS` stores the parameters of every car in `XML` files, so we had access to the information about the car we were trying to simulate. It was the default car: `car1-trb1`. We have decided to use the *bicycle model*.

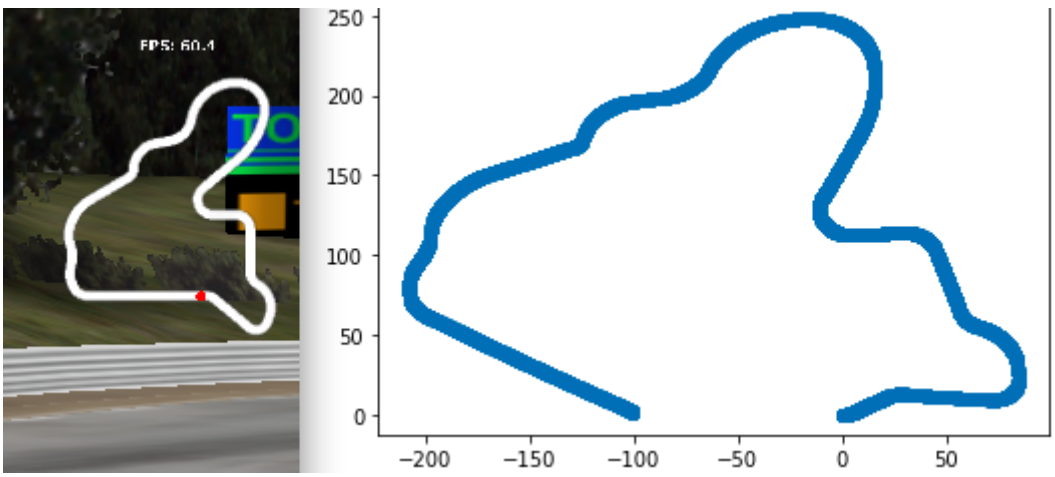
A brief description of it:

- the model clamps both front and rear wheels into one
- it finds the center of rotation based on the speed, length and steering angle of the car.
- rotates the car around found center using the angle between the car direction and x axis of the world (calculated using position from previous iteratiom)

Before actually moving the car we approximated its speed after pressing the gas or brake pedal with given power. For this we used multiple parameters such as:

- current RPM of the engine to take its power and torque into account
- current gear of the car to take its efficiency into account
- current fuel of the car in litres to add to mass of the car
- rolling resistance, inertia, radius of wheels
- braking disc radius, braking pad force and maximal brake pressure

After setting the starting position to $(0, 0)$ and mirroring every move of the car on the first lap with our model this is an example of the recorded track:



Here we figured that *MCTS* wouldn't have much time to return an action so the simulation depth wouldn't be big. We thought that if the simulation tree isn't deep then this model would return values very similar to real ones. We also thought that if position of our artificial simulated car will be calculated by slightly incorrect model but simulated the same way then the results might be valid.

Evaluation

For the evaluation we used the smallest distance between car position and point of recorded track and also distance of the car from start line. We didn't wanted the car to constantly steer so to limit its slalom we set the distance to the track to 0 if it was smaller than some value.

Results and conclusion

With beginning of the second lap we started the *MCTS* algorithm on parallel thread with **0.3s** to return an action. We set the simulation depth to $\texttt{k} = 10$. We restrained the action space to $\texttt{steer} \in \{-0.3, -0.2, ..., 0.3\}$; $\texttt{gas}, \texttt{brake} \in \{0.3, 0.6\}$ and to drive only on the 1st gear. We also made sure that gas and brake are not being pressed at the same time. It achieved an average of **12000** iterations during the given time. The results were very unsatisfying, the car 'wobbled' for a few metres and the turned into the wall most of the times.

2.4 Empirical attempt

After failure of our previous model we decided to fully change the approach. We resigned from using just input avaible only to the default client and we used position of the car from the game. This instantly gave us perfectly recorded tracks from the first lap. We tried the previous simulation system on this track but we didn't notice it act differently so we proceeded to change the simulation.

Simulation

This time we wanted it to be simple and fast. We calculated direction vector of the car by taking position previously used as root of the tree and the current one and normalizing it. Then we rotated it by `current_steer` * `max_angle` * `S_const`, where `max_angle` = 0.366519 *rad* and `S_const` = 1.3. `S_const` was designated empirically. Let's call this vector `newDir`. Then we calculated `addVec` - vector we were adding to the current position.

- if the action was to press gas pedal and `acc` was the given pressure then:

$$\texttt{addVec} = \texttt{accMult} * \texttt{acc} * \texttt{newDir},$$

  where $\texttt{accMult} = 9.1$ was designated empirically.

- if the action was to press brake pedal and `brake` was the given pressure then:

$$\texttt{addVec} = \texttt{brakeTransfer}(\texttt{brake}) * \texttt{newDir},$$

  where $\texttt{brakeTransfer}(\texttt{x}) = -0.8 * \texttt{x} + 1.5$ was designated empirically

Evaluation

We evaluated the states the same way as before.

Results

The algorithm also worked the same as before on the same action space and achieved about the same number of iteration per given time of $0.3s$. This time after tuning the parameters the car drove as intended - it tried to hold on to the center of the track and could easily drive a full lap. But this was still only on the first gear so the car didn't need to brake for the whole lap to finish it. When we allowed the car to drive on 3rd gear it had too much speed on the curves and it slid when trying to turn - our naive simulation have not considered losing grip.

2.5 Further work and observations

Here we listed changes that we also tried but didn't work:

- limiting action space to `steer`, `brake`, `gas` $\in \{-0.1, 0.0, 0.1\}$ and adding the value to previously saved action when applying to a state. We think it didn't work because the algorithm couldn't keep up with the changes actually happening in the game. We tried lowering the amount of time given to the *MCTS* but then the number of iterations were to small to come up with a meaningful action
- We tried adding value of the state for every state during rollout in *MCTS* and extending the depth of simulations but it didn't improve the driver. Probably because the simulations were inaccurate on the curves on higher speeds so the algorithm didn't know its out of its track bounds.
- We also tried using source code of the game to simulate the car. We tried cloning car that was passed to the server and simulate it in parallel to the existing car. It caused collisions with exising car and in general didn't work out.

3 Hypersonic

After mediocre results in `TORCS` and shortage of time, we have decided to move into another project idea - to write a bot for `Hypersonic` game on **CodinGame**. `Hypersonic` is a *classic Bomberman-style* game with up to 4 players. We are controlling a player whose target is to destroy as many boxes as possible or to eliminate other players with its bombs. Player can collect bomb range boosters and additional bombs that drop from different boxes on the map. Placed bomb returns to player's inventory once it explodes. The game is played in turns - all players perform their moves simultaneously and every agent has 100 milliseconds for one move (except 1000 milliseconds for the first move in the game).

3.1 First attempts

In order to get to know the game mechanics, we have written a simple rule based bot - it used search algorithm to calculate distances to boxes in the game and acted accordingly. This bot ended up in the **Bronze League**.

3.2 Further work

To be able to develope more complex agents we had to write properly working game engine, which could simulate the course of the game. For that engine to be useful for agents, evaluation function which evalutes the state of the game had to be also developed. Although every agent could have slightly different evaluation function, all of these functions ended up looking very similar and followed the same rules, as state quality should be objective.

3.3 Game engine and evaluation function

Evaluation function took into the account the amount of boxes agent managed to destroy with more pressure on the boxes destroyed in the near future. This was important as we want to destroy boxes as soon as possible - not only to be able to destroy further boxes, but also to be sure we will be the one who destroys them - as we go into the future, our moves become more and more uncertain as we have decided not to simulate other agents because it was computionally infeasible. Also from the tests we have performed, it was clear, especially at the beginning of the game, to focus on our agent more than on other players.

Other important part of the evaluation function was to encourage the agent to place the bomb in the first place - we have to remember that bomb explodes in 8 seconds - that is in 8 turns of the game, so connecting cause with delayed result was critical. In order to do that explosion map was

computed - for every cell, information about time in which the cell will explode and who will explode it. Boxes which agent will explode in the future were rewarded proportional to the time we have to wait for that box to be destroyed.

Last part of the evaluation function was to reward for picking up boost items. As we start with just 1 bomb with 3 range, out possibilities are pretty limited and the game has slow pace at the beginning, so it is important to pick up boosts to be able to destroy more and more boxes. From tests and experience it was better to reward extra bomb items more - but still up to the certain point.

3.4 Beam Search

After the first solution we have decided to try something more sophisticated - the *Beam Search* algorithm.

Main algorithm

At every iteration of the algorithm `BEAM_WIDTH` best states were kept. For every one of them we have generated all of its descendants which were part of the next iteration of states. If more than `BEAM_WIDTH` descendants were generated in total, only `BEAM_WIDTH` best were kept. After all the iterations the first action of the best individual was returned.

The main algorithm ended up in the **Silver League** - it had many problems as it was pretty slow (game engine and the algorithm itself needed optimization), *Beam Search* spent a lot of computional power to consider states which led to death after some amount of time (e.g. bomb explodes in 8 seconds).

Further improvements

If we were not been able to survive in the generated descendant, that state was automatically ignored (we were not able to survive if in the next 8 turns, every possible cell in which we could be would blow up, assmuming no other bombs would be placed). This survivial mechanism was also used in further algorithms tested by us as it was filtered out losing states very quickly (it as also used as a part of the algorithm specific evaluation function). Additionally *Point Beam Search* mechanism was used - every descendant was classified into the cell in which our player landed. If more than `POINT_BEAM_WIDTH` descendants were generated for the given cell, surplus were abandoned.

This modifications led to the considerable improvement of our agent's behaviour as it was able to get to the **Legend League** - around **140th** place in the ranking (out of around 380 participants).

In order to further improve the quality of our algorithm, both the game engine and the algorithm were optimized leading to approximately 3x speedup. From the game analysis of our agent it was observed that it often led itself to death by placing the bomb while the other player simulataneously placed the bomb in such a configuration, which led our agent to no escape situation - it was the main source of all the deaths of our agent. This was taken care of by computing the set of possible actions in the initial state under assumption that every other player will place its bomb (if has any) where he stands in the next turn. In the first iteration of the algorithm only possible actions were considered. That mechanism lead to the great avoidance of no escape situation - it was effectively one look ahead mechanism under pessimistic asssumption.

This improved agent got to the very top of the **Legend League** - **8th** place in the ranking.

Beam Search agent's final parameters (mentioned above):

`BEAM_WIDTH`	**200**
`BEAM_DEPTH`	**15**
`LOCAL_BEAM_WIDTH`	**20**

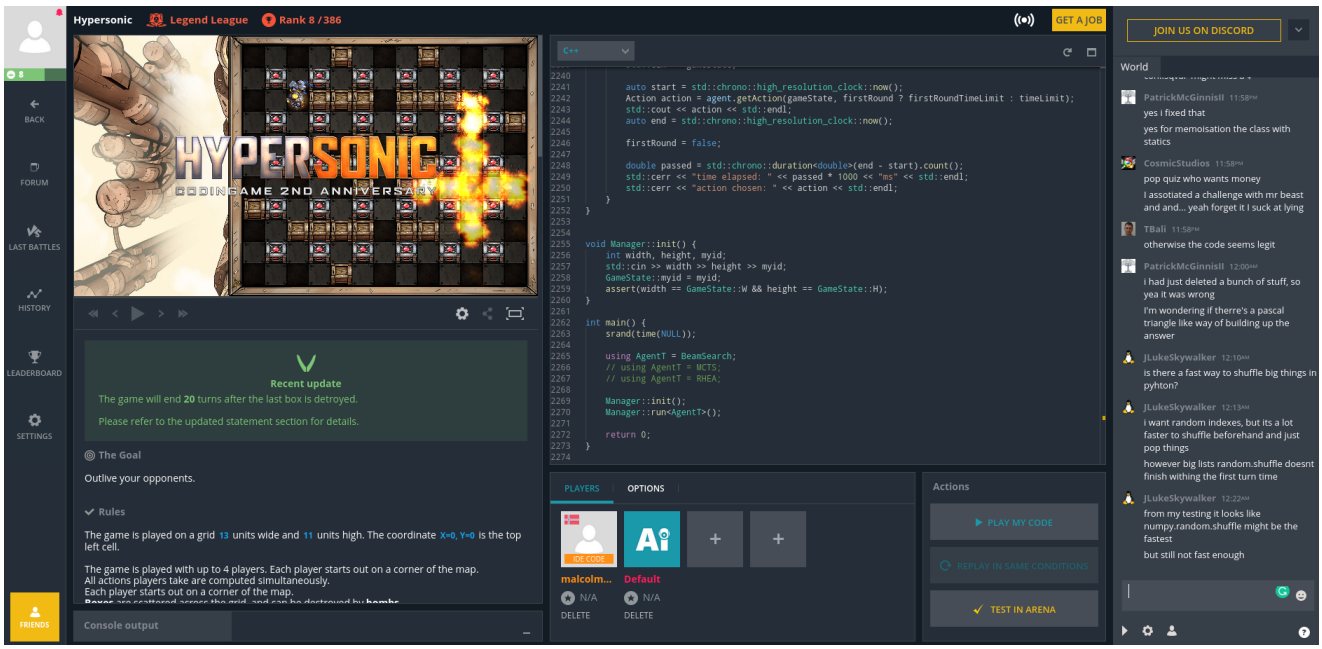


Figure 4: Proof of *Beam Search* agent's ranking place on the **CodinGame** platform.

3.5 Rolling Horizon Evolutionary Algorithm

We have also decided to try radically different approach to the problem. *Rolling Horzion Evolutionary Algorithm* (*RHEA*) is the algorithm which tries to find the best sequence of actions for our agent, up to some point (`HORIZON_LENGTH` next actions to be exact). To do that we will use evolutionary algorithm with standard evolutionary operators.

Main algorithm

In each iteration we will maintain population of `POPULATION_SIZE` chromosomes, each of length of `HORIZON_LENGTH` genes. We will perform selection of parents from the current population which will generate `OFFSPRING_SIZE` children using crossover operators. Children will be mutated with `MUTATION_PROBABILITY`. At the end we will create the next population from children and current population.

Roadblocks

One problem with creating sequences of moves is that not every sequence of moves is valid - we could have wall ahead. This is especially true when we apply our evolutionary operators which could modify a part of the chromosome (sequence of actions). Other problem is that performing an action could lead us to death, which could be easily avoidable. This led us to the solution in which we ignore actions which are invalid - either because we die or action performed is technically invalid.

Chromosome evaluation

Chromosome (sequence of actions) was evaluated by performing actions it containts in sequence and returning evaluation of the state in which we end up. If the state in which we end up was no-escape sitatution, evaluation was decreased by some amount, depending on how far in the future the state was not survivable (the further in the future, the smaller the punishment was).

Evolutionary operators

First we have to mention that action was represented as position delta in two directions from the current position of the agent.

Crossover As a crossover operator we have considered *uniform* crossover and *shuffle* crossover operators. Both crossover functions produced two children from two parents. In *uniform* crossover, one split point was randomly chosen and children were created by merging appropriate parts of two parents. In *shuffle* crossover, every gene was randomly chosen from one of the parents (not chosen one was distributed to the other child). Performed tests indicated that *uniform* crossover was much better than *shuffle* crossover. One explanation could be that *uniform* crossover is very natural when it comes to problems related to path finding (which our game has a lot of in common).

Mutation We mutate every gene of every child with `MUTATION_PROBABILITY` probability to a randomly chosen gene (random action).

Selection

Parents for recombination are selected using *roulette wheel method* - depending on chromosome fitness value, it has different probability to be chosen to create offspring.

Replacement

We have considered two methods of replacement of current population to the new one - $(\mu + \lambda)$ or (μ, λ). In the first method we create next population with the best individuals from the current population and children. In the second method we create next population with the best individuals only from children. Both methods were tested in the **CodinGame** and our own environment and we concluded that $(\mu + \lambda)$ replacement is much more efficient as it keeps previously found good solutions, which could have not been created by our evolutionary operators, as some fraction of genes in child could be wasted as they could be invalid in the current state.

Testing

We have created our own environment in which we could set up a battle between different agents and optimize parameters. One important indication if *RHEA* algorithm works correctly is if population diversity is mantained (the main problem of evolutionary algorithms). Previously mentioned parameters and operators, were tweaked not only for the agent to perform as good as possible but also to maintain population diversity, which is especially true if we want to find good solution.

Results

When tested in the **CodinGame** environment, agent ended up in the **Gold League** - **1st** place in the ranking - directly below the boss. It was clear that we could improve something to achieve better results, as noone should be forced to end up directly below the boss and not advance to the next league.

Improvements

In order to further improve the algorithm, individual chromosome evaluations were reduced (which was the main cost in *RHEA* iteration). This led to 2x speed up, which could be used to perform more iterations of the algorithm. Additionally similar mechanism of possible actions was introduced into the *RHEA* algorithm - first actions of all individuals could be only from the set of possible actions in the current state under assumption that every other player will place its bomb in the next turn.

Final RHEA results

Those improvements greatly enhanced the behaviour of *RHEA* agent as it ended up in the **Legend League** - around **170th** place (380 participants in total).

RHEA agent's final parameters (mentioned above):

POPULATION_SIZE	**100**
OFFSPRING_SIZE	**100**
HORIZON_LENGTH	**8**
MUTATION_PROBABILITY	**0.6**

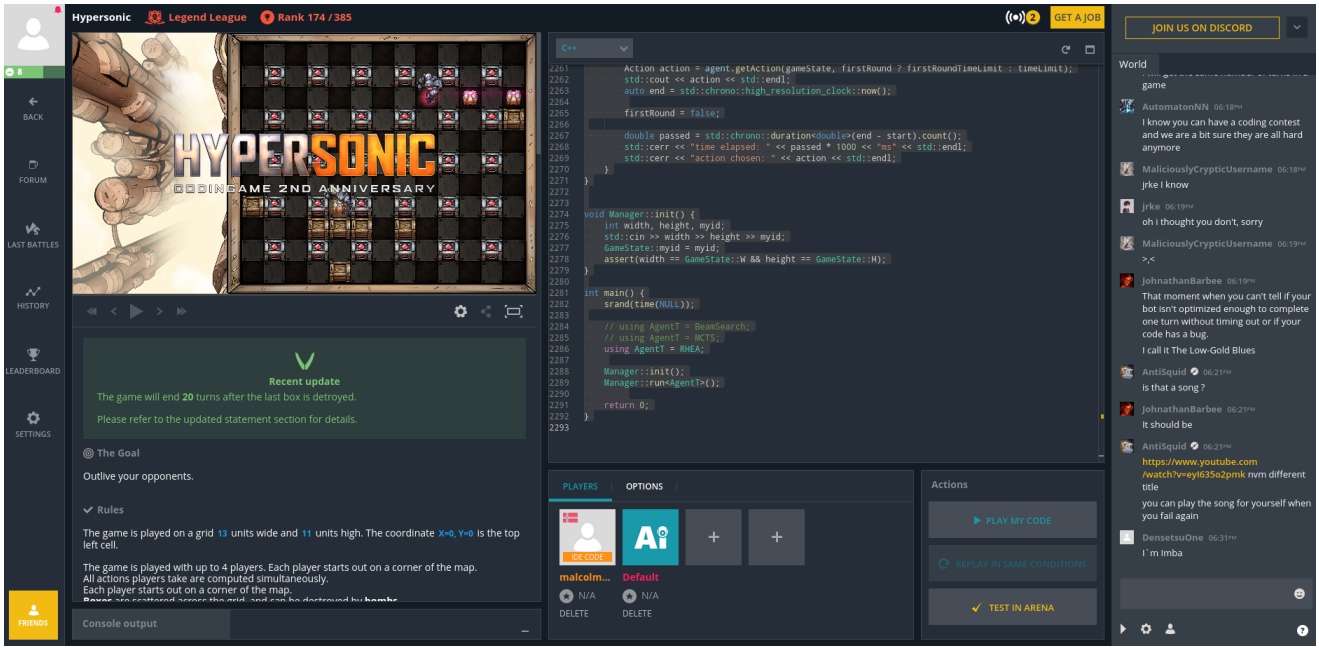


Figure 5: Proof of *RHEA* agent's ranking place on the **CodinGame** platform.

3.6 MCTS

We decided to also try *MCTS* for this problem. We went for the single player variant of the algorithm as the game was too long to simulate whole and we kept scores based on the evaluation function of game states.

Main Algorithm

On every iteration we created new root of the tree based on the current game state. The algorithm performed for $0.1s$ on every iteration ($1.0s$ on first) and then returned action from node with best average score.

Traverse

We used *Single Player Monte Carlo Tree Search* UTC formula for selecting the child:

$$\overline{X} + C \cdot \sqrt{\frac{\ln t(N)}{t(N_i)}} + \sqrt{\frac{\sum x^2 - t(N_i) \cdot \overline{X}^2 + D}{t(N_i)}}$$

where:

$$\begin{aligned} \bar{X} &- \text{ average reward of state} \\ N &- \text{ visits of current node} \\ N_i &- \text{ visits of i-th child} \\ C &= 1000.0 \\ D &= 10000.0 \end{aligned}$$

Rollout

In rollout we were simulating course of the game for $\texttt{k} = 8$ next turns only for the player. Our policy for action selection was random. If the player reached state when he desintegrated $\texttt{deathReward} = -10.0$ was returned and evaluation value of reached state in other case.

Evaluation

The evaluation was the same as in *Beam Search* algorithm described earlier.

Results

This algorithm has reached **10th** place in the **Silver League**.

3.7 Agents comparison

Comparison of the agents presented above in different configurations were performed using our custom game environment. Results are as follows:

- *Beam Search* vs *RHEA* - 82% vs 18% winrates
- *Beam Search* vs *MCTS* - 99% vs 1% winrates
- *RHEA* vs *MCTS* - 95% vs 5% winrates
- *Beam Search* vs *RHEA* vs *MCTS* - 75% vs 21% vs 4% winrates

4 Conclusions

Considering results, tests and analysis in the presented report, *Beam Search* agent performed the best, resulting in not only reasonably behaving agent but also in the one that ended up in the very top of the ranking of the hightest league on **CodinGame**. The second best agent was *RHEA* agent, which performed noticeably worse than *Beam Search* agent, but still ended up in the **Legend League**. *MCTS* agent performed the worst, which is probably due to the lack of time to properly test and improve the algorithm and tweak its parameters.