Performing in contests @codingame

I had **so much fun** in taking part of all the codingame challenges! It's a real pleasure each time to discover the new subject, start thinking about my bot strategy and climb the leaderboard with the time I can spend on it!

I participated in all of them (except Platinum Rift 2) and I'm pretty proud of the results: I **won** the winamax open day, and I am often in the **top 1%** always in the **top 5%** with a usual invested time around 15 hours per challenge.

So it's time to **share**, best practices I learned, things that work, things that don't work, but also a set of **tools and code** you might find useful. Enjoy!

Code management

Git is your friend

You certainly know already git, and may want to skip this part.

But if you don't, git is a source code manager that allows you to get back and forth between **different versions** of your code. What's super cool is that you can create **development branches**, starting from a stable base and develop a **new feature**. I would strongly recommend you to create a new branch per feature so that you can easily merge your code in your reference branch. If your development is not great, you will be then able to directly get back to your previous version without having to backup any file. You can also easily work **in parallel** on several developments and make them advance easily.

It needs a little bit of practice, but I could not engage a contest without it.

Finding Ideas to implement

Do not reinvent the wheel

We are all the same: you hear this little voice in your head telling you "Code! Code! Code! Code!"? **STOP**! Refrain your coding drive! You can have fun during a few minutes coding the first ideas, but if you want to reach the top, at some point you will have to **search on the web** if someone has not already written something that could be useful.

Of course you can have a look at the subject by itself. I used the google AI challenge tron **winner post mortem** and extracted the things that could have been applied to tron, I read interesting (but not enough efficient) **articles** on Quoridor which was close to "The Great Escape", but that's usually not enough. You should also read interesting articles related to the **artificial intelligence field** and **game theory** that will greatly help you during a contest. You will even find some interesting free **MOOC** on the web on the subject and I would recommend that you take them outside of the contests so that you can learn without the contest pressure.

And don't hesitate to read all the **post mortem** of previous contests players: it is a great source of excellent ideas you will be able to re-use later on!

Your best friend is not your IDE, your best friend is your pen and your paper

I always take some time to write things on **paper**. It allows me to step back and I am usually happy with the outcome. **Drawing things** usually help because it is there you can solve equations, discover

characteristics etc. For example, when dealing with collisions on "Poker Chip Race", writing the vector positions, the speed vector, the circle and putting all that together help me find an efficient way to compute it very quickly. On "Smash the Code", I realized that when evaluating a grid, there was no groups of 4 cells and that to count the groups of 3, it was enough to count the number of cells with two neighbors of the same color etc. It could be frustrating as this time could be spent coding, and usually the list of ideas is long, but this **step back** is in my opinion mandatory.

If you have an **idea, just note it** in a centralized place. It will help you when selecting the next feature to implement ☺

Watch the other bots

Might seems obvious, but the **other players** can have **super ideas** too. Watching the other bots play, and train your bot against them is a huge source of ideas. You can always have access to the complete leaderboard, and you can watch the replay of the best bots. You will certainly have ideas from the way they play. In "Platinum Rift" I remember seeing SaiskyApo dumping all his pod in Antartica during the first turn in 4 players games, and was impressed by the effect on his ranking. It took me a few minutes to implement, and was a huge improvement in the leaderboard: I was never losing 4 player games anymore and was often ranked second!

When training against other bots, **select bots** that are a **little bit higher** than you, but not the top ones you might never beat. It will help you see if your new version is slightly better or not. You can also change the opponent because your strategy might be more or less efficient depending on the other strategy.

Selecting the best features

You will have dozens of ideas, some are good, some are in fact bad. Here is some advice to select the **bests**. Anyway an idea might be good later on, don't hesitate to write them somewhere.

Best return on investment first

If you have several things to implement, it is not always easy to select what you should code first. In fact, over time I realized that you should evaluate the **return on investment** and select the feature that has the **best ratio**. It might seem obvious, but let's try an example: on one hand you have a super feature that will cost you 2 hours of dev, and might help you gain 50 places, and another feature which will take you ten minutes, and might gain 10 places. Then code the ten minutes feature first! You will be able to measure the enhancements more quickly, you will be able to develop the other feature during that time, and you never known, your two hours' feature is perhaps super bad in fact.

Learn to kill your "genius" ideas

Often you will have some ideas to implement a new feature which looks great. It takes time to develop, but you are sure, it's a killer idea, you will ruin all the opponents!! The problem is that you will usually find issues when you will watch your bot, potentially have enhancements ideas, and iterate over time. If I have an advice, when you feel you can't make it, fix yourself a **fixed amount of time** at the end of which you will simply **drop the idea** and try to concentrate on something else.

For example, in "Coders Strike Back", I spent 5 days trying to find the optimal trajectory and speed considering there is no opponent to go as fast as possible on the track. I got into complex stuff, trying to define an optimal turn radius, optimal speed and so on. It was way too long, and I was aware of it!! I fixed the deadline, but felt I was so close I continued a little and it was a **mistake**. I ended up coding a very poor defensive strategy in a few minutes (which made me gain something like a hundred

places… ROI first you remember??), and ran clearly out of time before I was able to implement any opponent strategy consideration…

Watch your bot play

You must **watch your bot play**, and there are several reasons. First you might "see" **bugs** that will be obvious when in real conditions. It might even break your "genius" ideas and you will see they have strong weaknesses other bots might exploit.

Check **why you are losing games**. Just ensure it is not a stupid technical issue such as a time out or a crash. It might ruin your results. Try to understand why you lost, and it is usually a good source of new features or enhancements to develop.

Dealing with all those "magic constants"

During the development, you might define a few **magic numbers**, particularly in evaluation functions, and let's face it, you defined them based on guess but nothing really serious. So of course you can try to submit your code with different values and try to see if your bot behaves better or not, but that's still trying to guess a good value, and the feedback loop is taking too much time!

If you are able to **test** the impact of these parameters **without submitting** your code, you can use a **genetic algorithm** to find out the best parameters. Example: in smash the code, I optimized the balance between the different parameters (number of block of 3, number of block of 2, etc) using the following criteria: the number of turns needed to reach a certain score threshold with a predefined set of randomly generated blocks. You have to be careful to evaluate your parameters in the same conditions, but when ready, you can learn a lot from there.

Note that this approach might allow you to train neural networks and other machine learning techniques that will cost you CPU on your computer, but will cost you almost nothing in your bot AI! I never had the chance to try, but if I can I'll do it!

Don't panic when running out of time

The **last day**, just before the deadline the situation evolves: more and more developers are submitting again their code which results in several effects:

1. The **global quality** of the bots **increases** strongly: everyone is placing its best submission, and since the deadline approach, the time invested is greater than before
2. More and more players are submitting their code which results in a lot **top bots** are under evaluation. All those bots are **climbing** the leaderboard, and does not appear at their real value anymore. It shifts artificially the other bots at a **sur-evaluated** ranking. Most of us knows that the rank in the Sunday of the deadline is usually better than the final results!
3. It is more difficult to evaluate if your submission is better or not because there are so many matches in parallel it will take **more time** to get a final **ranking**

So in the final hours I concentrate **most of my efforts** on watching the last replay of my bot looking for strange behaviors or **bugs**, enhancing global **performances**. But never developing features with a strong impact: I would have difficulties to ensure it is really enhancing the performances of the bot against the others. So if I have an advice: in the final hours don't make something stupid, you had a week to work on your bot, it's not in the last minutes you will change it ☺.

Testing

Rely on tested code

You know what? When I heard about the first challenge I did (google AI challenge "Planet Wars" in 2010!) I took this as an opportunity to practice **TDD** (Test Driven Development). But applying it to an AI challenge is not so easy. You identify quickly several parts in your AI: parsing the game state, the game logic as described in the rules, and your strategy implementation by itself. The point is that your **strategy might evolve quickly** and you don't really know in advance what is a good play or what is wrong. It really depends on your strategy, and most of the test cases you will write might be broken when you change a few settings.

So I tried a different approach. During Planet Wars I tried to do only what you would call system tests. Meaning that I was giving the full game state to my bot placed in obvious situations and asserting it was producing the correct results. The problem was that the tests were going through all the bot, and it was really hard to test little adjustments. Moreover, it prevents you from testing non obvious test cases because when your bot logic evolves, it might react differently. I discovered a bug later on in one of the basics of the game engine, and really regretted I wasn't able to discover it more quickly.

Now I usually test more small pieces, and put more or less effort on each part of the bot:

- **Game parsing code** is usually **covered quickly**. It's easy to test, and won't change over time.
- **Game logic** must be **very well covered**. It won't change over time and a bug here would kill your bot efficiency. If you should only test one part, it is this one.
- **AI algorithm** (minimax, genetic, monte carlo etc) should also be **well tested**. The advantage is that you might import them from previous contests, to be able to start with something that works well. On the other hand if there is a bug here, your bot will likely play very badly, and you will see it quickly.
- **High level AI logic** might evolve a lot and change quickly. It is also hard to test because it relies on other layer results. So I usually don't really test this part but put **some log** to confirm it works as expected.
- I always **secure** a few **key situations** by doing some **end to end tests**, giving the entire game state to my bot, and asserting the behavior. But the result should be obvious else you will have to adapt it when your AI evolves

Tests really represent a **safety net** with an almost instantaneous feedback. I am often very happy that when coding something new, some red tests appear and just shows me directly I broke one of the key behavior I secured previously. It allows me to analyze why it has been broken and potentially abandon the idea or enhance it. When I won the "Winamax Open Day", I started by adapting my tests to the new rules, then adapting the code quickly and adding tests for one or two new key features. That's it!

Logs usage

Each turn you have the chance to print to the System.err what you want. You are the only one who can read it, and you should **definitely use it**. You can use it as a debug tool but I really prefer writing unit tests for that. Here is what I like to print:

- Some **general statistics** related to the game. They can be the position of each players, the number of remaining turns, the number of pods I have etc. They are usually some key properties to make choices in the AI later on such as attack or defend but that are not available by default in the game inputs.

- The **high level AI strategy** chosen from the previous parameters. I am attacking, I am defending etc.
- Some **performances indicators** such as the computation time, the number of simulations done etc.
- Some **code** that represent the **game state** directly translated in Java. That's super great to debug or add some new system tests: you just have to copy paste this code in a new unit test and you can debug very quickly what's going on and why your bot is behaving like this during this turn.

Measure the improvements without committing the code

It is always frustrating to wait for a new submission to stabilize, and even then, the **Trueskill** system used to rank your AI is only giving an **estimation** of the skill so if you don't have a significant improvement in your ranking, you can't really know if your new version is **better or not**.

During a period, I was re-implementing the **game engine**, and was locally playing matches between my different versions of bot. I was then able to evaluate if a new version was better than the others using the Trueskill ranking system. But I faced two issues:

- The **time required** to re-implement the game engine and package it correctly takes too much time. On the 8 days long contest, you end up in starting your AI after 3 or 4 days and you quickly run out of time
- If at the beginning it is really efficient because you implement new features that are really increasing the skills of your bot, in the end you might produce a bot which is **just efficient against your previous bot versions**, and not really efficient against the others. In particular, all your bots will have the same weaknesses, and a new version that exploits it will just get worst results.

So now I try to identify a **part of the game** I want to optimize and compare the different versions on it. For example, in coders strike back, I optimized the driving logic of my bot considering a single pod in several tracks. I measured how many steps it required to do several turns and then compared the results of several strategies selecting in the end the one that produced the best results on various tracks

Tooling

Competitive programming on gihub

A few days ago I shared a java project on github: competitive-programming. This project has been designed to **share** some **code in Java** with the community having in mind the constraints we have when doing online challenges: no external library, and ending up with a single file.

This project contains mainly two parts:

1. A builder allowing you to **spread your code** across several classes in **separated files**. This is one of the clear prerequisite to be able to share easily some code from a challenge to another.
2. A set of **classes**, that are enough **generic** to be used **from a challenge to another**. Those classes covers for example the following areas:
    a. Time management
    b. Game theory algorithm (Minimax, MaxNTree etc)
    c. Graph scans (BFS)
    d. Genetic algorithm

e. Vectorial operations
f. Disk physics management (collisions, movements)
g. …

Don’t hesitate to contribute and to issue pull requests!

Conclusion

I don’t believe there is only one way to do well in contests. All I’ve written above are my **personal preferences**, and as in most of the cases, you should deal with it in a **pick and choose** mode. I hope that you’ve found it useful and that you learned something. Because in the end, remember, if you are not competing for a job, it means you are here for the **fun**! So **enjoy**!!

See you on the 25th June for the next challenge: codebusters!

Grégory “Manwe” Ribéron