PokerChipRace Manwe's strategy

Overview :

The core of the AI is quite simple. I have a class Proximities that allow me to compute all the positions and speeds of all the chips in the incoming 5 turns if everyone is waiting. It gives me all the reachable targets that could be touched in these 5 turns, how many accelerations are required, and when should I start accelerating. More on this in the dedicated part.
A first step is to guess the other player moves. So I make them play as if I was playing myself, considering no-one is moving. When I have their orders, I apply them, simulate again all the positions for the upcoming 5 turns and I play myself.
For any player, I just go over his chips, and decide what to do for each of them.
First I double check if the chip is still alive in five turns. If it is not the case, I double check who is killing it, and try to avoid this oil droplet/chip.
Else I am looking for a target. I take all the reachable chips/oil droplet, filter the ones that are not interesting or dangerous and rank them. To rank them I use the gain of mass they will give me, taking into consideration the number of impulsions I will have to do to reach them and also taking into consideration the other objects that will be in this area. It allows the chips to prefer an area where there is several small droplets than just a little bigger one alone. When ranked, I select the best, and depending if I should start accelerate now or later, either I aim at it directly, either I wait.
This is it ! Let's go into the details:

Proximities

I won't go in details on the physics implementation, see the physics thread on this subject in the forum. The difficulty has been to have something that was correctly managing absorbing/bouncing/ex-pulsing without taking too much time. I did not split the simulation of a turn in 500 sub steps, but computed collision times, and applied a simple reordering of the collisions to treat the first ones first. I had something quite close to what the engine was really computing, so I have all the positions of the chip for the coming 5 turns
In order to select the targets and potential dangers, I introduced the notion of “proximities”. When you accelerate your chip, you change its speed, and so on the next turn, you might be able to reach any point that is in a circle around your position up to 200/14 units. You will be able to touch any object (if it is not accelerating itself) if the distance between your two center, minus the sum of your radius is bellow 200/14. The second turn, if you accelerate now, you can reach all the points up to 400/14 units. But on the second turn, you might accelerate again, and in this case reach all the points up to 600/14 units. If you wait on the first turn, but accelerate on the second, you can reach any point up to 200/14 units. With this logic, depending on the distance of the other chip/oil droplet, and the turn in the future you are analyzing, you will deduce for each target if it might be reached, and if yes, how many acceleration it will require and when should be the first acceleration.
Distances were precomputed at the initialization in my case, and you can have the following table to represent the different distances reachable in function of the turn you are simulating, and the accelerations you might make :

	1 acceleration	2 accelerations	3 accelerations	4 accelerations	5 accelerations
1 turn	200/14	X	X	X	X
2 turns	2x200/14	3x200/14	X	X	X
3 turns	3x200/14	5x200/14	6x200/14	X	X
4 turns	4x200/14	7x200/14	9x200/14	10x200/14	X
5 turns	5x200/14	9x200/14	12x200/14	14x200/14	15x200/14

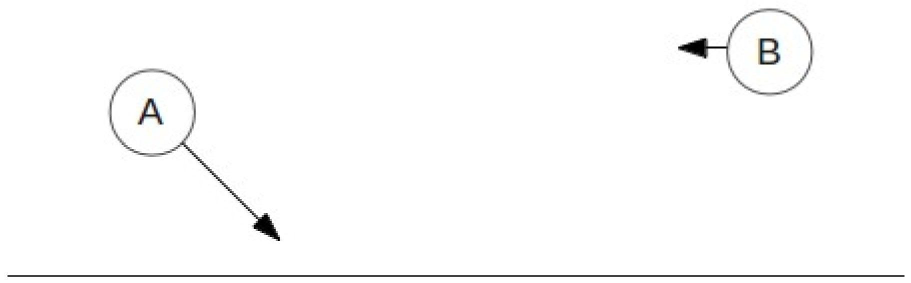
At each turn, when I computed the distance between the two items (distance includes the radius), several possibilities might be correct. In this case, I choose the one that require the less accelerations and that require to accelerate the later possible.
For example if at the 3rd turn, I have a chip that is at 200/14, I'll consider I can reach it with one acceleration and wait two turns before I have to accelerate.

I also precomputed the loss of radius for each acceleration, so I can quickly know if I might be able to absorb the target after making all the accelerations and take into account when subtracting the sum of the radius.

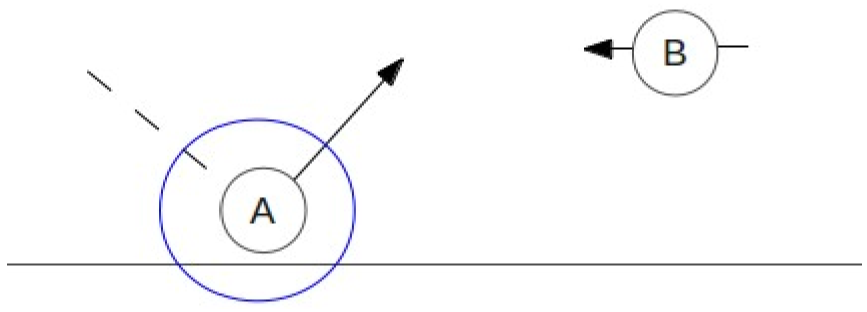
OK so on each turn I can say if a target is reachable, and if yes, how many acceleration it will require and in which turn I will have to accelerate. But as I am computing positions for each of the next 5 turns, I keep only the best proximity for each couple of chip/oil droplet. I choose again the one that require the less accelerations and that require to accelerate the later possible.

Let's take an example !

Here are two chips A and B. they are moving each turn from the length of the arrow which represent their speed :

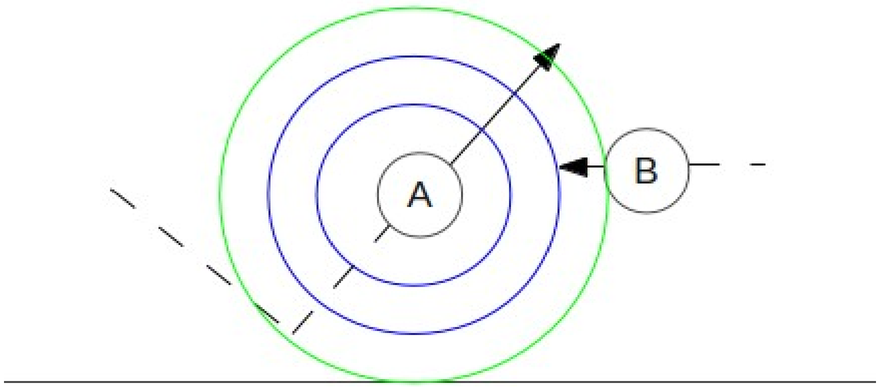


OK so on next turn, A could have been chosen to move in an direction, and so might be positioned on the blue circle :



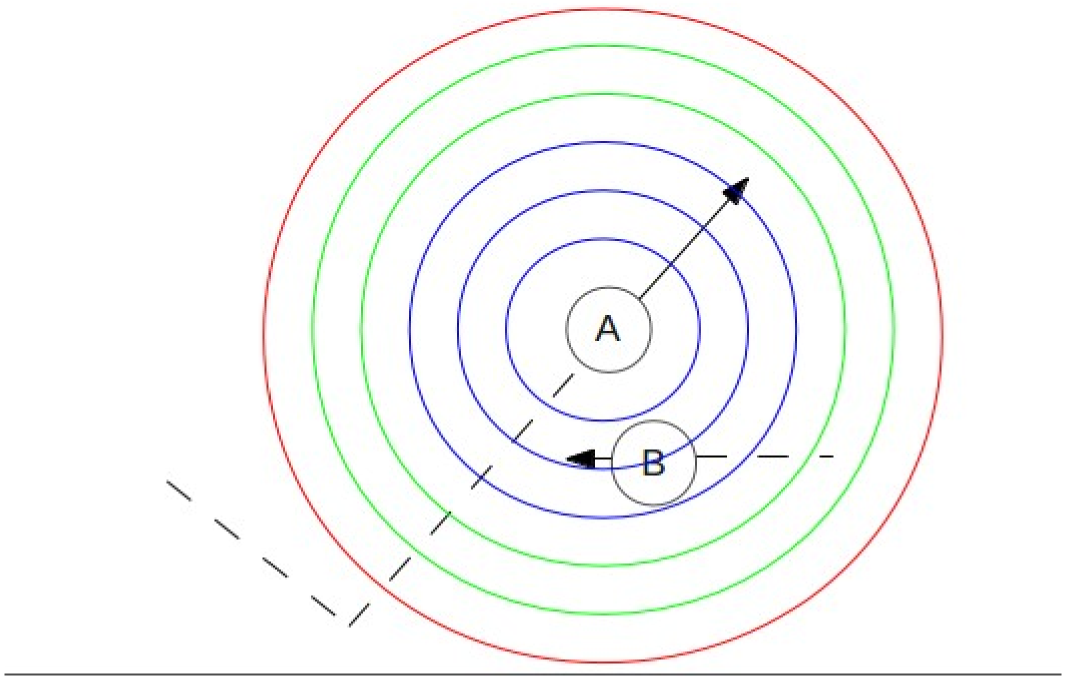
A can't reach B, there is no Proximity that exists between those two.

Let's see on next turn :



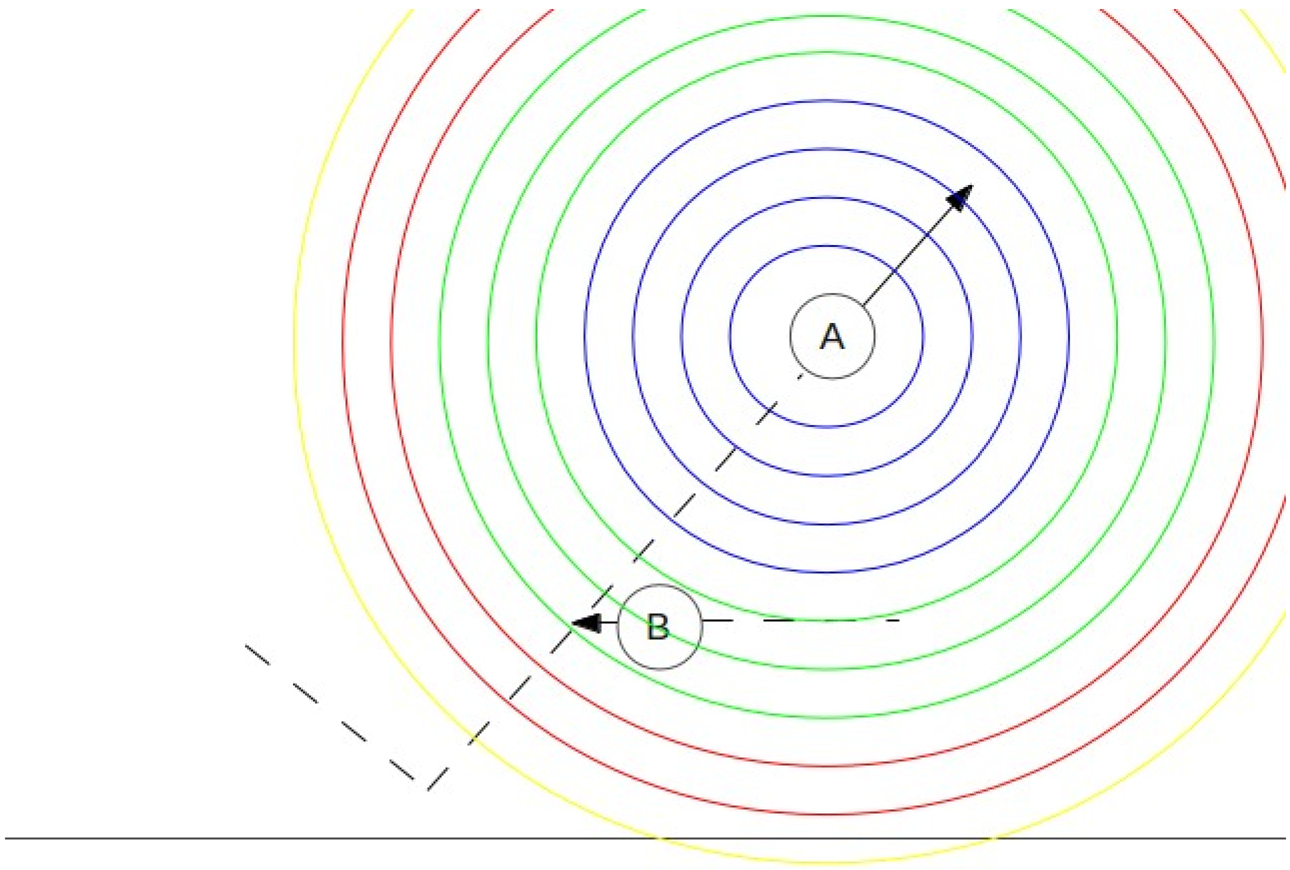
A can reach the first blue circle by waiting a turn, then making an acceleration. But it can also reach the second circle by making one acceleration and wait on the second. Finally it can reach the green circle by accelerating twice.

As B is touching the green circle, A has a way to collide with B on turn 2 if it accelerate twice successively. Great we have a proximity between those two ! Let's memorize it as the best proximity for now. Let's see what's happening on next turn :



As you understood, blue circles represent one acceleration on turns 2,1,0, green are double accelerations starting from turn 1,0, and red is a triple acceleration. Good news, the best combination we have at turn 3 is to wait the first turn, and then accelerate only once. As it is better than what we have at turn two, we keep this proximity for the A,B couple.

OK and on turn 4 ?



The best we have is to wait one turn, and make a double acceleration. This won't be better than the situation we have at turn 3, so we will keep the turn 3 as the good solution. Turn 5 won't be better than turn 3 too.

When calculating next turn positions, if two chips collides, I memorize their merging position, it will be helpful in the next chapter.

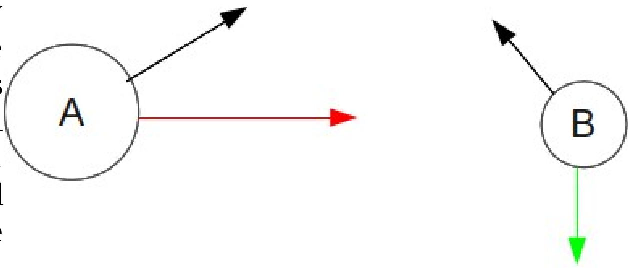
Avoiding death

Once I simulated the next five turns, I double check if my chip is still alive. If not, I retrieve which chip/oil droplet killed it, and if it is not an allied, I try to avoid the collision.
I didn't find a really great effective way to avoid the collisions. I tried different approach to determine in which direction should I accelerate :

- take the perpendicular of the relative speed at the collision time. As there is two possible directions, I take the one that is not making me cross the path of the chips.
- Take the direction opposed to the position of the collision
- Compute the direction in which I will throw my ejected droplet in the absorber in order to try to slow it

In fact all have their advantages and might work better in some circumstances. I did not have time to completely try to mix them, but I feel like there is something to do right here. After several tries on the leader-board, I kept the first approach, knowing that if I am close to a wall or if the absorber is accelerating I won't certainly be able to avoid it.

Here is an illustration of the solution I kept. My B chip will be absorbed by A. A and B are moving according to the black arrows representing their speed. I first compute A relative speed to B represented by the red arrow. Then I take the perpendicular direction and select the one that is not going in direction of the collision point represented by the green arrow



Target selection

If my chip is not going to die, I try to find the best target it can absorb. I scan all the proximities that have been extracted during the next turns simulations and I first filter the target that match one of those criteria :

- I can't absorb the target : depending on the number of accelerations I have to make, my radius will be inferior or equal to the target
- The target is way too small : It won't even compensate the loss I will have to do to go in its direction.
- The target is hidden by another bigger object that will make me suicide if I try to reach it.

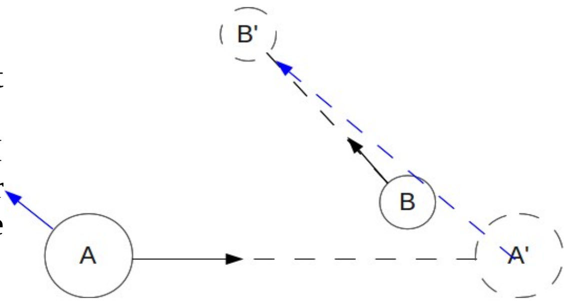
Then I sort the candidates computing a score for each of them. For each target I start by computing the mass gain it will represent. Then considering the collision position, I scan quickly which will be the other chip/oil droplet that will be in a circle of two accelerations over two turns. For each of them, I subtract their mass if the mass is more important than the sum of my absorber chip and the target, and in contrary I add half of the mass if the regroupment could absorb it. I am pretty sure it could be enhanced, those constants are « magic numbers », but I did not have time to work on it.
Then I divide the mass gain per the number of turns in which the collision is possible to finally select the target that will have the best adjusted mass gain per turn.

If no target is eligible, I just wait. There is certainly things to do instead of waiting right ? But did not have the time to work on it too...
If to reach the target I need to wait and to accelerate at a later turn, I just wait too.

Aiming

OK so in case I found a suitable target and I have to accelerate this turn, I have to decide how I should go in its direction. If more than one propulsion is necessary, I choose to simplify the computation (even if it is not so complex when you can solve the single case) and just take the direction in which the target will be on the collision turn.
Example bellow :

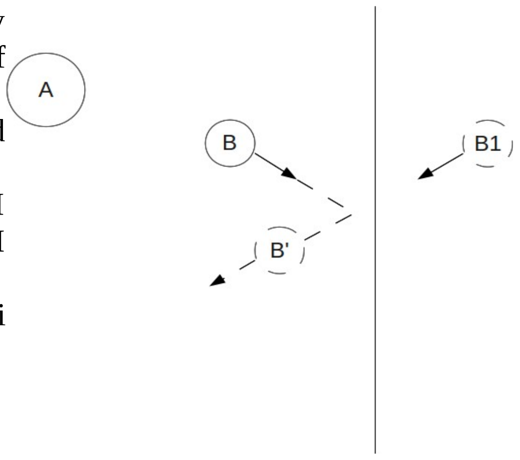
I am A, and want to go reach B in 3 turns, knowing it will require two propulsions.
In three turns, A will be in position A', B in B'. So I accelerate in the same direction that the A'B' vector right now, and ask to go in the direction of the blue vector.



Now what if it require only one acceleration ? In the beginning I was missing targets, because if the speeds are much more important than your radius, you might just miss the fast targets if you aim at a turn position like I did when multiple propulsions are required. So you have to compute the time of collision **t** where you will reach exactly the target. In fact I computed the case where the two circles positions are at the same coordinates. But wait, in order to do this computation, you have to consider the position and speed of each target as if they where moving in right lines during all the turns. So you have to compute the position that they would have at your current turn taking into consideration their speed and position at the collision turn. This is the key to take into consideration wall bouncing.

Let's take an example again :

Still with A and B, A wanting to absorb B. It greatly simplify to consider only the relative speed and position of B to solve the problem.
Let's consider A is not moving for now. What is the good direction to go in order to absorb B ?
I know that the collision should be possible at turn 3, so I consider the position on turn 3 of B which is B'. Then I subtract 3 times the B' speed vector which gives B1.
Starting from this B1 position, I have to find a vector Vi where :



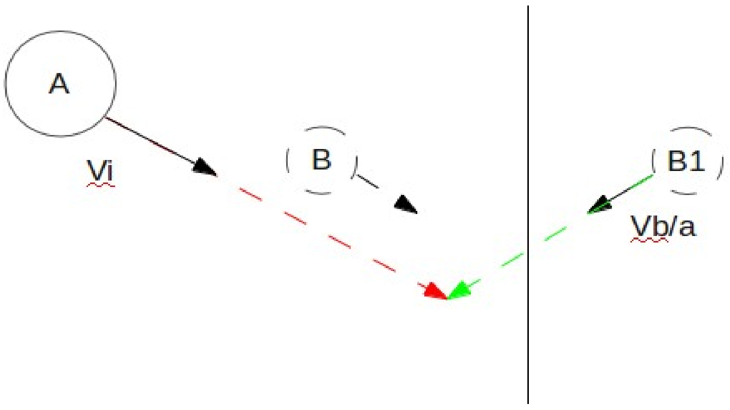
$$k \times \vec{Vi} = k \times V\vec{B1} + A\vec{B1} \quad \text{and} \quad \|\vec{Vi}\| = \frac{200}{14}$$

finding k ends up in a second degree equation (back to high school!!)

As you know there might be no real solution, and in this case I take the real part of the solution. Else I take the lowest positive k if there is two solutions.

k found I can determine the exact collision point where both centers will have the same coordinates.

It should end up accelerating the A chip in the red arrow direction :



Notes

As for the previous challenges, I had to decide which language I will use. I choose Java over C++ because I wasn't sure that performances should be a limitation, over more that now I know C++ is not executed with optimizations. Even with two simulations of five turns, my answers are still taking around 50 ms maximum. It also allowed me to write unit test more easily (~40 test cases), catching bugs really often and ensuring my bot wasn't breaking the previously developed features.