Π-KIMONO

An AI that plays Pikomino

Authors
Hugo Passe and Arthur Vinciguerra
ENS de Lyon
November 2023

Contents

1 One turn game

1.1 Model

To compute the optimal policy in the one turn game we use a Markov decision process. The way we chose to represent it is the following:
The actions are the information of

- *dice* : The dice to pick.
- *stop* : A boolean, True if the player wants to stop, False otherwise.

A state is the information of

- *dices* : The dice launch, a tuple of eight or less elements. The tuple is sorted in increasing order.
- *picked* : The numbers already kept, a tuple of size at most 6, also sorted.
- *score* : The current score.
- *stop* : A boolean, True if the player decided to stop, False otherwise.

The worm face is represented by 0 while the other faces are simply represented by their value. Moreover we can reduce the number of states by regrouping the states that have *stop* = True into a few states : If *S = (dices, picked, score, True)* is a state, then it can be replaced by :

- *S = ((),(),score,True)* if *picked* does not contains 0.
- *S = ((),(0),score,True)* if *picked* contains 0.

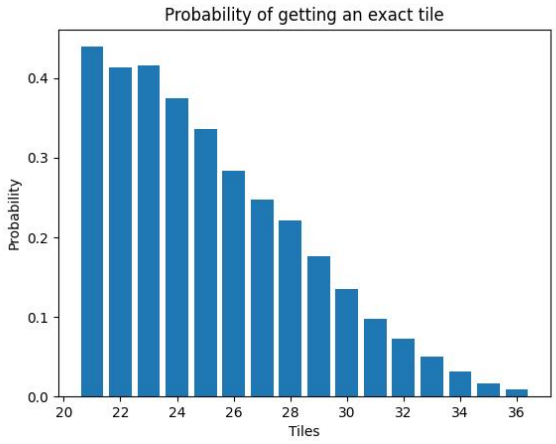
Since the score is bounded by 40, we only have 80 terminal states. With this representation we can compute our 68648 accessible states using hash tables in approximately 45s.

1.2 Computing the probability matrix

Given a state *s = (dices,picked,score,stop)* we can easily compute the probability $P(s, a, s')$ and find all the states s' for which $P(s, a, s') \neq 0$. For the variables *picked*, *score* and *stop* there is not choice and the computations are obvious. We only have to compute the probability of a dice launch up to a permutation. For a N faces sorted dice launch $d = (a_1, \ldots, a_n)$, we denote by $k_i(d) = \#\{a_k < i | k \in \{1, \ldots, N\}\}$ for $i \in \{1, \ldots, N\}$. The the number $N_{\text{perm}}(d)$ of permutations of the dice launch d is,

$$N_{\text{perm}}(d) = \prod_{i=1}^{N-1} \binom{k_{i+1}(d) - k_i(d)}{k_i(d)} \tag{1}$$

Then the probability for a launch of n dices of N faces to be equal to d up to a permutation is $\left(\frac{1}{N}\right)^n N_{\text{perm}}(d)$.



1.3 Value iteration

We showed how to compute all the states and the probability under any possible action to go into any other state. Therefore we pre-compute this data and store it in hash tables. Then we can do policy iteration to solve Bellman equation for all accessible state s,

$$w(s) = \max_a \ r(s,a) + \sum_{s' \in S} P(s,a,s')w(s'). \tag{2}$$

To illustrate the policy we give some actions computed under

1.4 Computing probabilities

Here we summarize the probabilities of achieving some goals, using the best policy to fulfill each of them. To do so we use a specific reward vector for each goal and use the law of total probability. For a goal G we have the following,

$$P(G) = \sum_{d \text{ dice launch}} P(d)P_d(G). \tag{3}$$

Using a vector that is 1 on scores that are in our goal and 0 on those that are not, we have that the policy's value in a state with dice launch d is the probability $P_d(G)$ of achieving the goal. The full probabilities are plotted in figure 1.4.

Goal	Reward vector	Probability of success
Pickomino of value 1 or more	(1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1)	0.893
Pickomino of value 2 or more	(0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1)	0.680
Pickomino of value 3 or more	(0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1)	0.346
Pickomino of value 4 or more	(0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1)	0.0868
Tile number 24	(0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0)	0.375
Tile number 27 or more	(0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1)	0.528

2 Stochastic Gradient Descent

To get an AI for a full game, we can implement the current player and repeat its strategy for all rounds. It basically is a greedy algorithm: we take short-term optimal strategies and hope they lead to long-term optimal strategies. Unfortunately, for this game, the greedy strategy does not seem optimal. We can realize that by playing against it (see instructions in the README file): it can choose to steal one of our tiles instead of picking the last available tile with fewer worms and winning the game.

2.1 Stochastic Gradient Descent Player

To have a better player, we parametrize the problem with (α, β) such that:

- The cost c is equal to α times the number of worms on the top tile that this player has (or 0 if the adversary has no tile).
- The reward vector r is such that $r_i = c$ if obtaining i does not provide any reward, $r_i = 2\beta W$ if obtaining exactly i allows the player to steal a tile with W worms from its opponent, and $r_i = W$ if obtaining i allows it to obtain a tile with W worms.

We consider $f(\alpha, \beta) = player(\alpha, \beta) - player(1, 1)$, where $player(\cdot, \cdot)$ is each player's number of points at the end of the game. Our goal is to find (α, β) that maximizes f. Basically, we want to find the one-turn player that beats the regular one-turn player by the most amount of point difference. Because f is a random variable, the idea is to make a gradient descent on its expectancy:

$$(\alpha_{n+1}, \beta_{n+1}) = (\alpha_n, \beta_n) + \eta_{n+1}\mathbb{E}(\nabla f).$$

We try to approximate $\mathbb{E}(\nabla f)$ with:

$$\mathbb{E}(\nabla f) \approx \left(\frac{\mathbb{E}[f(\alpha + \varepsilon, \beta)] - \mathbb{E}[f(\alpha, \beta)]}{\varepsilon}, \frac{\mathbb{E}[f(\alpha, \beta + \varepsilon)] - \mathbb{E}[f(\alpha, \beta)]}{\varepsilon} \right).$$

And we get the expectancies of each f by taking the mean of m games. We take the following parameters:

- $(\alpha_0, \beta_0) = (1, 1)$
- $\varepsilon = 0.01$
- $\eta_n = \frac{0.02}{n^{0.6}}$
- $m = 100$

These parameters were chosen to counterbalance the big variance of f. For computational reasons, we have decided for this part to reduce the number of dice to 4 and the tile set to $[11, 18]$. Every iteration takes approximately 10 minutes and we therefore only have a partial answer to the question in fig 1. Unfortunately, the random noise due to the randomness of each game seems a bit too big to make the gradient descent work.

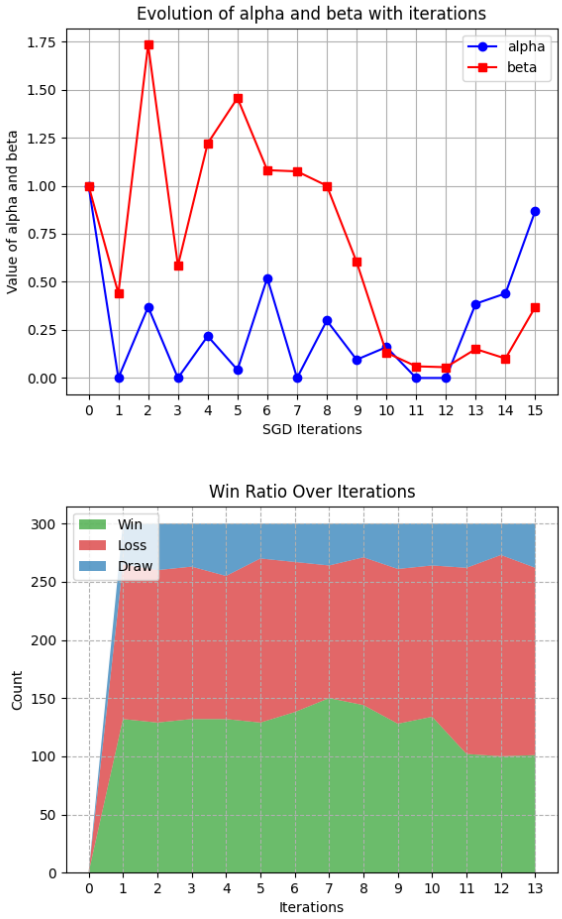


Figure 1: Evolution of the SGD player against the number of iterations

3 Difficulties

The major issues we ran into while carrying out this project were due to our choice of coding it in Python. In fact, we realized a bit too late that we did not know how to make optimized programs in Python, a major issue considering the number of computations the project needs. Because of this, we had to use only 4 dice for the gradient descent, and it was still very slow because of the amount of games we had to play to evaluate the gradient. We also tried to do a multithreaded SGD computation but Python does not seem to handle it very well and strange bugs started to make the code crash.