Baby Tetris First Part Report

ERGIN Seçkin Yağmur and GROUSSON Dylan

January 6, 2026

https://github.com/diaarca/baby-tetris/

Contents

1 Question 1: Discounted MDP For Player

The objective of this part is to choose the optimal policy that maximizes the expected cumulative discounted reward over time for the "player".

States
The state is represented by the current configuration of the grid, and the next piece to be placed.
$S :=$ (current grid, next piece to place)

Actions
The actions correspond to a position and orientation to place the piece on the grid.
$A :=$ (position, orientation)

Transition Function
The transition function is defined as $P(s'|s,a) := 1/2$
There are two possible pieces (I or L) as the next piece to place, and considering our state representation, and our assumption of uniform distribution between the pieces, each transition has an equal probability of 1/2.

Reward Function and Discount Factor
The reward function is computed based on the number of lines cleared after placing the piece, and a discount factor λ is used to weigh future rewards.
$R(s,a,s') :=$ coefficient * number of lines cleared with $\lambda \in [0,1]$

1.1 Finding Optimal Policy

We use the value iteration method to find the optimal policy that maximizes the expected rewards, iteratively updating the value function for each state based on the Bellman equation until it converges to the optimal value (margin ϵ).

$$V(s) = \max_{a \in A} \sum_{s' \in S} P(s'|s,a) \cdot (R(s,a,s') + \lambda V(s'))$$

Value iteration is better suited for maximizing the expected discounted reward since it computes the optimal value then extracts the policy and it's easier to implement.

1.2 Implementation

1.2.1 Main Architecture

Based on the our design choices from Section 1, we decided to use an object oriented architecture (C++) for our implementation.

For our structure we have declared 7 classes:

1. *Point*: classical point (x,y) class used for grid coordinates
2. *Field*: contains a boolean grid s.t. grid[i][j] = 1 if the corresponding cell is filled in the field
3. *Tromino*: corresponds to the baby Tetris pieces (I Piece or L Piece)
4. *State*: contains both a field grid and an incoming Tromino
5. *Action*: represents a choosable action on a given state (combination for placement and rotation of an incoming Tromino)
6. *Game*: contains the configuration (completed lines scores), the current state and score of a baby Tetris game (the configuration is loaded from a file : *config.txt*)
7. *MDP*: encompasses all the necessary methods in order to compute the value iteration over a game

Now that we presented the overview of the global implementation architecture, we will describe our implementation choices.

1.2.2 Implementation Choices

- **Pieces teleported to their final position**: In order to reduce the number of intermediate states when choosing an action, we compute on a state all available actions, then according to the policy, we make a choice over these actions and we teleport the incoming piece in the current state directly to its final position (with the corresponding rotation)
- **Reward Function**: As defined in Section 1, the reward function takes both the current state s and an available action a_s. Since we can calculate the available actions, we know in which state s' we're going to arrive and we can compute directly in s' the number of completed lines before removing them from the game. Then we abstract the reward function by counting the number of complete lines in the resulting state
- **Game Ending**: According to the subject, the game should end whenever a piece exceeds the height limit of the grid. Since we compute all available actions on a given state, if a piece cannot be placed on the grid, there will be no available action for that state, the game will terminate when there is no available action for the current state.
- **Max Iteration**: Since the value iteration isn't an exact method, if we fix an ϵ too small, we might iterate for a very long time. For this reason, we fix a maximum number of iteration for the expected value improvement to have a value iteration method that computes in reasonable time, even if the expected value vector is less qualitative than the ϵ margin we fixed

1.3 Results

Here, we will present multiple examples of execution of our implementation. In these executions we will use fixed values for: width = 4, $\epsilon = 0.00000001$, maxIteration = 100, probaIPiece = 0.5 and maxGameAction = 10000. Note that we fixed a small ϵ in order to approach the optimal policy, keeping in mind that the algorithm is bounded in iteration by the $maxIteration$. And the variant variables will be: the height $\in \{4, 5\}$ and the discount factor $\lambda \in [0, 1]$.

For each execution we will measure:

1. the number of iterations for the value iteration algorithm
2. the average expected gain of the optimal policy
3. the final score either at the end of the game or when we reach the maximum number of actions
4. the execution time of the said execution

1.3.1 Executions

```
All constants:
width = 4, height = 4, probaIPiece
= 0.5, maxGameAction = 10000
epsilon = 1e-08, maxIteration =
100, lambda = 0.1

i = 0 and delta = 6
i = 1 and delta = 0.6
i = 2 and delta = 0.06
i = 3 and delta = 0.00525
i = 4 and delta = 0.00045
i = 5 and delta = 0

average over final V 0.412926
....
....
**..
**..

Game Over! Global score: 7849 in
10000 actions
make run  3.02s user 0.08s system
92% cpu 3.340 total
```

height: 4 and λ: 0.1

```
All constants:
width = 4, height = 4, probaIPiece
= 0.5, maxGameAction = 10000
epsilon = 1e-08, maxIteration =
100, lambda = 0.4

i = 0 and delta = 6
i = 1 and delta = 2.4
i = 2 and delta = 0.96
i = 3 and delta = 0.336
i = 4 and delta = 0.1152
i = 5 and delta = 0

average over final V 0.473608
***.
**..
*.**
*.**

Game Over! Global score: 225 in
285 actions
make run  2.93s user 0.08s system
93% cpu 3.237 total
```

height: 4 and λ: 0.4

```
All constants:
width = 4, height = 5, probaIPiece
= 0.5, maxGameAction = 10000
epsilon = 1e-08, maxIteration =
100, lambda = 0.1

i = 0 and delta = 6
i = 1 and delta = 0.6
i = 2 and delta = 0.06
i = 3 and delta = 0.006
i = 4 and delta = 0.00058125
i = 5 and delta = 3.1875e-05
i = 6 and delta = 0

average over final V 0.462044
....
....
....
...*
.***

Game Over! Global score: 7864 in
10000 actions
make run  58.26s user 0.32s system
99% cpu 58.837 total
```

height: 5 and λ: 0.1

```
All constants:
width = 4, height = 5, probaIPiece
= 0.5, maxGameAction = 10000
epsilon = 1e-08, maxIteration =
100, lambda = 0.9

i = 0 and delta = 6
i = 1 and delta = 5.4
i = 2 and delta = 4.86
i = 3 and delta = 4.374
i = 4 and delta = 3.81358
i = 5 and delta = 1.88219
i = 6 and delta = 0

average over final V 0.676855
....
....
....
..*.
.***

Game Over! Global score: 8513 in
10000 actions
make run  58.20s user 0.34s system
99% cpu 58.809 total
```

height: 5 and λ: 0.9

1.3.2 Observations

Over our executions, we remark that the λ has a huge impact on the optimal policy computed. On the smaller grid (4×4), it is really difficult to reach the *maxGameAction* with $\lambda = 0.4$, since the model will prioritize short term decisions in order to increase its score faster. In comparison, with $\lambda = 0.1$, the model will not gain that much per action, prioritizing safer actions that allow a higher score on the long term.

That result was as expected, since in a 4×4 grid, the space for the risky moves is limited. The study of the game with the 4×5 grid shows that bigger grid provides a larger space to take risks in order to increase the global score. The game with $height = 5$ and $\lambda = 0.9$ illustrates this perfectly because it scores higher than the game with $height = 5$ and $\lambda = 0.1$, since the risky actions can be better expolited on a bigger grid.

In both cases, the total computational time is pretty acceptable (max 1 minute) even if it can be largely optimized (see Section **??**). About the exact same improvement, the average expected gain V cannot be really discussed here since there is a huge part of states that aren't reachable in practice.

Finally, we notice that the value iteration algorithm choice described in Section 1 was pretty relevant because we can observe the quick convergence to the optimal policy (at the last iteration, Δ is always equal to 0).

1.4 Reachable States

Instead of computing the value iteration on all possible states, we focus on the reachable states from a given starting state s_0. To achieve this, we implement a Breadth First Search (BFS) algorithm that explores all reachable states iteratively from the current state.:

Algorithm 1 Generate Reachable States

```
1: Q: a queue of states
2: s: a state
3: H: a hashMap of states
4: s ← s_0
5: Q.push(s)
6: while Q.isNotEmpty() do
7:     s ← Q.pop()
8:     for a: availableActions(s) do
9:         s_Placed ← applyAction(s, a)
10:        s_After ← s_Placed.completeLines()
11:        if Q.doesNotContain(s_After) and H.doesNotContain(s_After) then
12:            Q.push(s_After)
13:        end if
14:    end for
15:    H.push(s)
16: end while
17: return H
```

Then we can repeat our experiments that produces results that are way better:

```
All constants:
width = 4, height = 4, probaIPiece
= 0.5, maxGameAction = 10000,
epsilon = 1e-08, maxIteration =
1000, lambda = 0.1

i = 0 and delta = 6.10288
i = 1 and delta = 0.352625
i = 2 and delta = 0.0250445
i = 3 and delta = 0.0010013
i = 4 and delta = 4.28143e-05
i = 5 and delta = 1.79071e-06
i = 6 and delta = 4.8676e-08
i = 7 and delta = 1.07541e-09

average over final V 0.565842
....
....
..**
..**

Game Over! Global score: 7632 in
10000 actions
---------------------------------
Executed in    7.69 secs
```

height: 4 and λ: 0.1

```
All constants:
width = 4, height = 4, probaIPiece
= 0.5, maxGameAction = 10000,
epsilon = 1e-08, maxIteration =
1000, lambda = 0.9

i = 0 and delta = 8.36976
i = 1 and delta = 5.32365
i = 2 and delta = 3.54512
i = 3 and delta = 2.10366
...
i = 87 and delta = 1.33224e-08
i = 88 and delta = 1.0733e-08
i = 89 and delta = 8.64692e-09

average over final V 4.15109
*...
*..*
*..*
**.*

Game Over! Global score: 11118 in
10000 actions
---------------------------------
Executed in   26.53 secs
```

height: 4 and λ: 0.9

```
All constants:
width = 4, height = 5, probaIPiece
= 0.5, maxGameAction = 10000,
epsilon = 1e-08, maxIteration =
1000, lambda = 0.1

i = 0 and delta = 6.20552
i = 1 and delta = 0.477762
i = 2 and delta = 0.0334845
i = 3 and delta = 0.00181086
i = 4 and delta = 6.29689e-05
i = 5 and delta = 2.6382e-06
i = 6 and delta = 1.32337e-07
i = 7 and delta = 6.63914e-09

average over final V 0.559813
....
....
....
*...
***.

Game Over! Global score: 7620 in
10000 actions
---------------------------------
Executed in   46.73 secs
```

height: 5 and λ: 0.1

```
All constants:
width = 4, height = 5, probaIPiece
= 0.5, maxGameAction = 10000,
epsilon = 1e-08, maxIteration =
1000, lambda = 0.9

i = 0 and delta = 9.7563
i = 1 and delta = 6.7369
i = 2 and delta = 4.43912
i = 3 and delta = 2.70431
...
i = 88 and delta = 1.44415e-08
i = 89 and delta = 1.16674e-08
i = 90 and delta = 9.42616e-09

average over final V 4.54237
...*
..**
.***
.***
.***

Game Over! Global score: 13305 in
10000 actions
---------------------------------
Executed in  402.82 secs
```

height: 5 and λ: 0.9

We can then remark that with this new implementation, the value iteration algorithm is quite slower both in terms of computation (per iteration) and in number of iteration. For instance, the 4×4 game with $\lambda = 0.4$ wasn't able to play all the 10000 actions with being blocked while we can now play it with a way more "risky" policy ($\lambda = 0.9$).

2 Question 2: Adversarial MDP For Opponent

The goal of this part is to design an adversary policy that selects tromino pieces for the game with the intention of minimizing the players score. For this policy we use once again value iteration, but this time we minimize the expected cumulative discounted reward over time from the available actions. The policy chooses the tromino piece that leads to the lowest expected score for the player. We have the same states (reachable states), actions, transition function and reward function as in Section 1, the only change is the value iteration equation. The player will still use the optimal policy computed in Section 1 to play the game.

2.0.1 Choosing the Tromino Piece

We considered two different heuristic approaches for choosing the tromino piece:

- **Lowest Maximal Expected Reward**: The policy chooses the tromino piece that leads to the lowest maximal expected reward for the player.

$$V(s) = \min_{t \in \{I,L\}} \max_{a \in A_t(s)} \sum_{s'} P(s'|s,a,t) \cdot (R(s,a,s') + \lambda V(s'))$$

- **Lowest Average Expected Reward**: The policy chooses the tromino piece that leads to the lowest average expected reward for the player.

$$V(s) = \min_{t \in \{I,L\}} \frac{1}{|A_t(s)|} \sum_{a \in A_t(s)} \sum_{s'} P(s'|s,a,t) \cdot (R(s,a,s') + \lambda V(s'))$$

Both approaches produce similar results, with no remarkable difference. However, compared to the random adversary policy, where the tromino pieces are chosen randomly, the tromino value iteration, for both approaches, successfully decreases the player's score.

2.1 Implementation

We add a new method in the *MDP* class that computes value iteration for the adversary policy. Both the *trominoValueIterationMinMax* and *trominoValueIterationMinAvg* methods share a common value iteration framework. They primarily differ in how the next tromino piece is chosen by the adversary, which in turn defines the update rule for the state value function $V(s)$. The common part involves an outer loop for value iteration convergence and an inner loop iterating through all reachable states via a map 'V'.

2.1.1 Lowest Maximal Expected Reward (*trominoValueIterationMinMax*)

In this approach, the adversary aims to minimize the maximum possible expected reward that the player can achieve. For each state, the algorithm computes the best outcome for the player for each potential next piece. The adversary then chooses the piece that leads to the lower of these two maximal rewards.

```
// ... (within the loop over all reachable states)
// for (auto& [currState, currValue] : V)

    double maxI = 0.0, maxL = 0.0;
    for (const Action& action : currState.getAvailableActions())
    {
        // This inner loop evaluates outcomes for both next-piece possibilities
        for (State& placedState : currState.genAllStatesFromAction(action))
        {
            State afterState = placedState.completeLines();
            double vAfter = V.at(afterState);
            double reward = 0.5 * (placedState.evaluate(config_) + lambda *
   vAfter);

            const Tromino* t = &afterState.getNextTromino();
            if (dynamic_cast<const LPiece*>(t))
            {
                maxL = std::max(maxL, reward);
            }
            else if (dynamic_cast<const IPiece*>(t))
            {
                maxI = std::max(maxI, reward);
            }
        }
    }
    // The new state value (vPrime) is the minimum of the two maximal rewards
    if (maxL < maxI)
    {
        T.insert_or_assign(currState.clone(), std::make_unique<LPiece>());
        vPrime = maxL;
    }
    else
    {
        T.insert_or_assign(currState.clone(), std::make_unique<IPiece>());
        vPrime = maxI;
    }
// ... (the rest of the loop updates delta and V[currState] with vPrime)
```

Listing 1: Lowest Maximal Expected Reward

2.1.2 Lowest Average Expected Reward (*trominoValueIterationMinAvg*)

In this variant, the adversary selects the next tromino ('I' or 'L') that results in the lowest average expected reward for the player, averaged over all available actions.

```
// ... (within the loop over all reachable states)
// for (auto& [currState, currValue] : V)

    double avgI = 0.0, avgL = 0.0;
    int nbActions = currState.getAvailableActions().size();
```

```
    for (const Action& action : currState.getAvailableActions())
    {
        // This inner loop evaluates outcomes for both next-piece possibilities
        for (State& placedState : currState.genAllStatesFromAction(action))
        {
            State afterState = placedState.completeLines();
            double vAfter = V.at(afterState);
            double reward = 0.5 * (placedState.evaluate(config_) + lambda *
   vAfter);

            const Tromino* t = &afterState.getNextTromino();
            if (dynamic_cast<const LPiece*>(t))
            {
                avgL += reward;
            }
            else if (dynamic_cast<const IPiece*>(t))
            {
                avgI += reward;
            }
        }
    }
    // Normalize the sums to get the average reward per action
    if (nbActions > 0) {
        avgI /= nbActions;
        avgL /= nbActions;
    }
    // The new state value (vPrime) is the minimum of the two average rewards
    if (avgL < avgI)
    {
        T.insert_or_assign(currState.clone(), std::make_unique<LPiece>());
        vPrime = avgL;
    }
    else
    {
        T.insert_or_assign(currState.clone(), std::make_unique<IPiece>());
        vPrime = avgI;
    }
// ... (the rest of the loop updates delta and V[currState] with vPrime)
```

Listing 2: Lowest Average Expected Reward

2.2 Executions

According to our implementation, we made the same executions in order to compare our adversary policies. Here we play the value iteration policy (from Section /refsec:question1) for the player on 3 adversary policies: random (based on *PROBA_I_PIECE*), Min-Max value iteration and Min-Avg iteration:

```
All constants:
width = 4, height = 4, probaIPiece
= 0.5, maxGameAction = 10000,
epsilon = 1e-08, maxIteration =
1000, action policy lambda = 0.9,
tromino policy lambda = 0.1 and
number of simulation = 50

Average results:
Random Adversary moves: 11086.3
Min Max Adversary Moves: 9999
Min Avg Adversary Moves: 10000


-----------------------------------
Executed in   35.04 secs
```

height: 4 and $\lambda_{tromino}$: 0.1

```
All constants:
width = 4, height = 4, probaIPiece
= 0.5, maxGameAction = 10000,
epsilon = 1e-08, maxIteration =
1000, action policy lambda = 0.9,
tromino policy lambda = 0.9 and
number of simulation = 50

Average results:
Random Adversary moves: 11091.7
Min Max Adversary Moves: 9999
Min Avg Adversary Moves: 10000


-----------------------------------
Executed in   40.04 secs
```

height: 4 and $\lambda_{tromino}$: 0.9

```
All constants:
width = 4, height = 5, probaIPiece
= 0.5, maxGameAction = 10000,
epsilon = 1e-08, maxIteration =
1000, action policy lambda = 0.9,
tromino policy lambda = 0.9 and
number of simulation = 50

Average results:
Random Adversary moves: 13285
Min Max Adversary Moves: 9996
Min Avg Adversary Moves: 9996


-----------------------------------
Executed in  630.78 secs
```

height: 5 and $\lambda_{tromino}$: 0.1

```
All constants:
width = 4, height = 5, probaIPiece
= 0.5, maxGameAction = 10000,
epsilon = 1e-08, maxIteration =
1000, action policy lambda = 0.9,
tromino policy lambda = 0.1 and
number of simulation = 50

Average results:
Random Adversary moves: 13292.5
Min Max Adversary Moves: 14994 /!\
Min Avg Adversary Moves: 9996


-----------------------------------
Executed in  537.57 secs
```

height: 5 and $\lambda_{tromino}$: 0.9

2.2.1 Observations

We can first observe that the score is not that much reduced by the adversary policies even if it is more noticeable in the 4×5 case. This is caused by the lack of possibilities, we can only choose between 2 pieces where the player may have to choose between way more actions.

Contrarily to the player value iteration we can remark that the λ parameter doesn't affect the score of the game. This is due to the fact that the score is minimized in that case and then the factor with which we multiply the $V(s)$ will not change anything.

3 Usage

Different constants can be adjusted by modifying the corresponding *#define* statements in *src/Tetris.cpp*, *hdr/MDP.h* and *hdr/State.h*.

In order to recompile and run the whole project, execute the following command:

```
make run
```

Unlike the textual output shown in Section 1.3.1, in the submitted version of the code, there is an exhaustive description of the played game (evolution of the state at each action) as follows:

```
....                .*..                    .*..
..*.                ***.                    ***.
***. --- LPiece --> ***. -- Completion -> ***.
***.                ***.                    ***.
Current score: 734

.*..                .***                    ....
***.                ****                    .***
***. --- LPiece --> ***. -- Completion -> ***.
***.                ***.                    ***.
Current score: 735

....                ***.                    ***.
.***                .***                    .***
***. --- IPiece --> ***. -- Completion -> ***.
***.                ***.                    ***.
Current score: 735
```