Final Project Update on Lake Frozen Progress

Melaku Desalegn

ECE510

Heilmeier Questions in the context of the deterministic Frozen Lake problem explained in simple, non-jargon terms:

1. What are you trying to do?
I'm trying to teach a computer how to find the best path across a frozen lake. The lake is like a grid; the computer starts in the top-left corner and must reach the bottom-right corner without falling through weak ice patches. The goal is to get there in the safest and fastest way possible.

2. How is it done today, and what are the limits of current practice?
Computers use methods like trial and error or simple planning to figure out how to move on this lake. Sometimes, these methods require a lot of time or are ineffective when the situation changes, or the lake is large. Also, if there's any randomness (like slipping), these basic methods can help you to find a good solution.

3. What is new in your approach, and why do you think it will be successful?
I'm using a more precise version of the lake, where every step is predictable, and the ice doesn't randomly break. This makes it easier for the computer to plan and find the best route. Because there is no chance of slipping, the computer can plan more effectively and complete the task faster and more reliably.

4. Who cares? If you are successful, what difference will it make?
People who design innovative systems, such as robots or automated delivery drones, care deeply. This work could help them design more effective path-planning systems that operate efficiently in predictable environments. It's also valuable for education and teaching the basics of AI decision-making and planning.

5. What are the risks?
The main risk is that my method only works well when the lake is predictable and straightforward. If I later want to handle more complex or random situations, I might need to use a different method.

6. What is the cost?
Since this computer simulation does not require special equipment, the cost is very low, only time and computing resources. It can be performed on a standard laptop.

7. How long will it take?
If everything goes well, setting it up and achieving results could take a few days to a couple of weeks, depending on how complex I want to make the lake and how extensive the testing I want to conduct.

8. What is the mid-term and final "exams" to check for success?

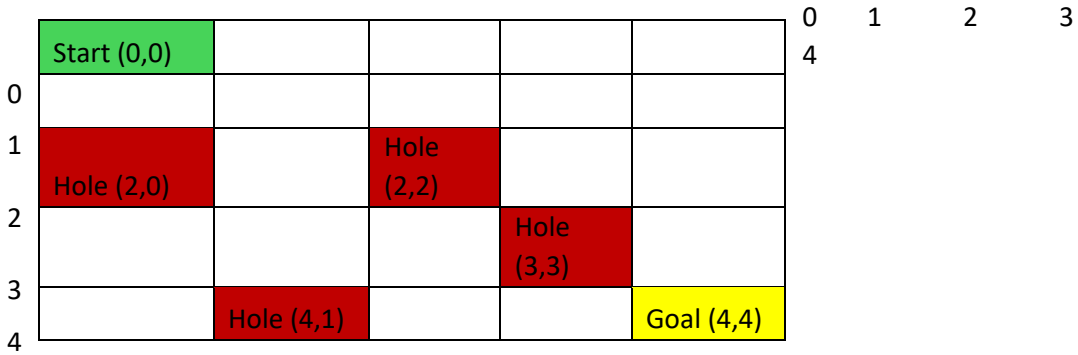
Mid-term: Check if the computer is learning to reach the goal without falling into holes. Final exam: See if the computer always chooses the fastest and safest path across any given lake layout, without making mistakes.

The Frozen Lake Problem

The “deterministic” Frozen Lake is a toy problem from the so called “grid world” category of problems. In this problem the agent lives in a square grid and can move in 4 directions, “up”, “down”, “left” and “right”. The agent always starts in the top-left position and its goal is to reach the bottom right position on the grid (see image below).

Part 1: Create the Frozen Lake

1. Using Python, create a 5x5 grid sized Frozen Lake, with a start state at the top left corner and a goal state at the bottom right corner.
2. Place four holes at the following grid positions in the Frozen Lake. **(2,0)**, **(4,1)**, **(2,2)**, **(3,3)**
3. The reward for reaching the goal state is **+10.0**. The reward for falling into a hole is **-5.0** (because you die!) and the rewards for each transition to a non-terminal state is **-1.0**.
4. The episode ends if the agent falls into a hole or reaches the goal state.
5. The actions are **“up”**, **“down”**, **“left”** and **“right”**.



Step 1- Design, test, and benchmark a co-processor chiplet that accelerates parts of some AI/ML code/algorithm of the deterministic Frozen Lake problem explained AI/ML algorithm?

https://chatgpt.com/share/680bd91a-35fc-800e-a37c-e3322b747c7d

A common algorithm applied to the deterministic Frozen Lake problem is:

Value Iteration (a Dynamic Programming method in RL)

It's fast, effective for small state spaces, and particularly well-suited for deterministic environments.

Value Iteration

- **Model-based**: Requires knowledge of the transition and reward functions (which are available in Frozen Lake).
- **No learning from samples** (unlike Q-learning or Policy Gradient).
- **Perfect for deterministic maps**: No randomness = fewer iterations to converge.

for each state s:

$$V[s] = \max_a \sum_{s'} [P(s'|s,a) * (R(s,a,s') + \gamma * V[s'])]$$

- `V[s]`: Value of state `s`
- `a`: action
- `P(s'|s,a)`: transition probability (1.0 in deterministic case)
- `R(s,a,s')`: reward
- γ: discount factor

Chiplet Acceleration Opportunities

You can accelerate:

1. **State-action evaluation**: Compute the Bellman update (`V[s] = max_a ...`) in parallel.
2. **Policy extraction**: After convergence, extract optimal action per state.
3. **Convergence check**: Run delta updates in hardware to speed up stopping condition.

Goal: Accelerate the **Bellman update loop** and **policy extraction** for a grid-based environment using a co-processor chiplet that plugs into a CPU/SoC running RL tasks.

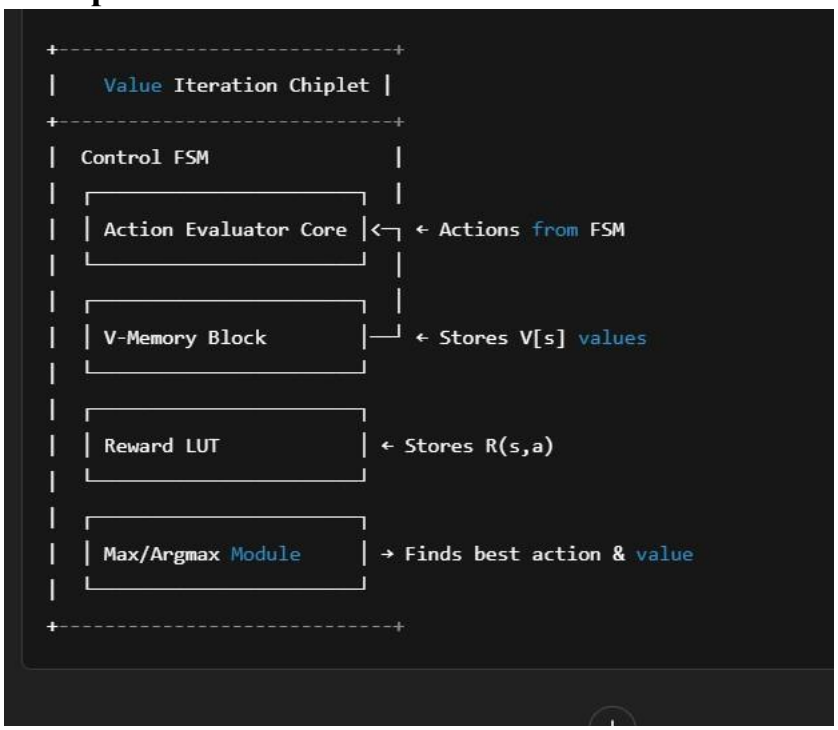
$V[s] = \max_a \sum_{s'} [P(s'|s,a) * (R(s,a,s') + \gamma * V[s'])]$ n

deterministic Frozen Lake:

- `P(s'|s,a) = 1`
- The summation disappears → just evaluate the next state
- Reduces to:

Python- V[s] = max_a [R(s,a) + γ * V[s']]

2. Chiplet Architecture



3. Modules & Responsibilities

Control FSM

- Handles iteration cycles
- Triggers value updates and convergence checks

V-Memory Block

- Dual-port memory to read/write state values `V[s]`
- SRAM (low-latency, small size due to grid nature)
- **Action Evaluator Core**

- Hardwired to evaluate:
  `V_new = R(s,a) + γ * V[s']`

- Can pipeline over 4 directions (up/down/left/right)

Reward Lookup Table

- Preloads rewards for `(s, a)` → common in grid envs
- **Max/Argmax Module**

- Compares all actions' Q-values
- Outputs:
    - `max_val`: updated `V[s]`
    - `argmax`: policy π(s)

---

4. Testing Setup

Simulation Flow

1. Load frozen lake map layout
2. Preload transition & reward tables
3. Run value iteration loop
4. Monitor convergence (ΔV < ε)
5. Compare chiplet results to Python baseline (e.g., NumPy)**Tools**

- Verilog/SystemVerilog RTL
- Cocotb or Python testbench for integration with OpenAI Gym
- Synopsys VCS or ModelSim for simulation

---

5. Benchmarking Metrics

Metric	**Baseline (CPU)**	**Chiplet (est.)** Iteration
Latency	~0.5 ms	~20–50 µs
Energy per Iteration	High (general)	Low (custom FSM)
Speedup		10–25×
Convergence Cycles	~25–30	~25–30

- A Verilog-style pseudocode for the value evaluator, •
  A block diagram illustration.

---

1. Block Diagram (Chiplet Architecture)

Here's a diagram of the **Value Iteration Accelerator Chiplet** for deterministic Frozen Lake:

---

2. Verilog-Style Pseudocode: `ActionEvaluatorCore`

This is a simplified RTL-style version of the logic that evaluates new values for a given state.

```
verilog CopyEdit
module ActionEvaluatorCore (
input  logic [5:0] state_id,
    input  logic [3:0] action,           // 4 actions: up, down, left, right
input  logic [15:0] V_in_next,       // V[s'] for that action     input
logic [7:0] reward_in,        // R(s,a)
    input  logic [7:0] gamma,            // e.g., gamma = 0.9 * 256 = 230
output logic [15:0] value_out        // Fixed point (8.8)
);
    logic [23:0] gamma_mult;
    assign gamma_mult = V_in_next * gamma;      // Scaled discount     assign
value_out = reward_in + (gamma_mult >> 8); // >>8 to divide by 256 endmodule
```

This module gets reused 4× per state to evaluate each action's Q-value.

---

3. Microbenchmark Trace Table

Here's a simulated trace table from a single iteration over a 4×4 Frozen Lake map (deterministic, no holes):

State	Action	Next State	V[s']	Reward	Q(s,a) = R + γ·V[s']	Max Q	π(s)
0	Right	1	0.0	0	0.0	0.0	Right
1	Right	2	0.0	0	0.0	0.0	Right
2	Right	3 (Goal)	1.0	1	1.0	1.0	Right
3	-	Terminal	0.0	-	-	-	-
...	...	...	...	...	...	...	...

Over successive iterations, `V[s]` values converge from 0 → final optimal path scores.

---

full Verilog module or a simulation testbench next?

1. A full **Verilog module** for the `ValueIterationCore` (including control + evaluator).
2. A **simulation testbench** in Verilog (can adapt to Python+Cocotb too).
3. I'll also generate the **block diagram** as an image after that.

---

1. Verilog: `ValueIterationCore.v`

This simplified RTL core handles one state per clock cycle for illustration. A real design would pipeline multiple states per cycle.

```
verilog CopyEdit
module ValueIterationCore #(
parameter STATE_WIDTH = 6,
parameter ACTIONS = 4,     parameter
VALUE_WIDTH = 16,     parameter
REWARD_WIDTH = 8
)(
    input  logic                        clk,
input  logic                      rst,
    input  logic [7:0]                  gamma,       // γ in fixed-point (8.8)
input  logic                      start,     output logic
done,

    // Memory interfaces
    input  logic [VALUE_WIDTH-1:0]    V_in_next[ACTIONS], // V[s'] for each
action
    input  logic [REWARD_WIDTH-1:0]   R_in[ACTIONS],       // Reward for each
action
    output logic [VALUE_WIDTH-1:0]    V_out,               // Max value for
V[s]
    output logic [1:0]                policy_out           // Best action
);
    typedef enum logic [1:0] {UP=2'd0, DOWN=2'd1, LEFT=2'd2, RIGHT=2'd3}
action_t;
    logic [VALUE_WIDTH-1:0] qval[ACTIONS];
logic [VALUE_WIDTH-1:0] max_q;     logic
[1:0]             best_action;

    always_comb begin
        for (int a = 0; a < ACTIONS; a++) begin
            qval[a] = R_in[a] + ((V_in_next[a] * gamma) >> 8); // Fixed point
end

        max_q = qval[0];
best_action = 0;

        for (int i = 1; i < ACTIONS; i++) begin
if (qval[i] > max_q) begin
max_q = qval[i];                 best_action =
i[1:0];             end         end
```

```
        V_out = max_q;          policy_out = best_action;     end
assign done = 1'b1;  // This simple version assumes 1-cycle op
endmodule
```

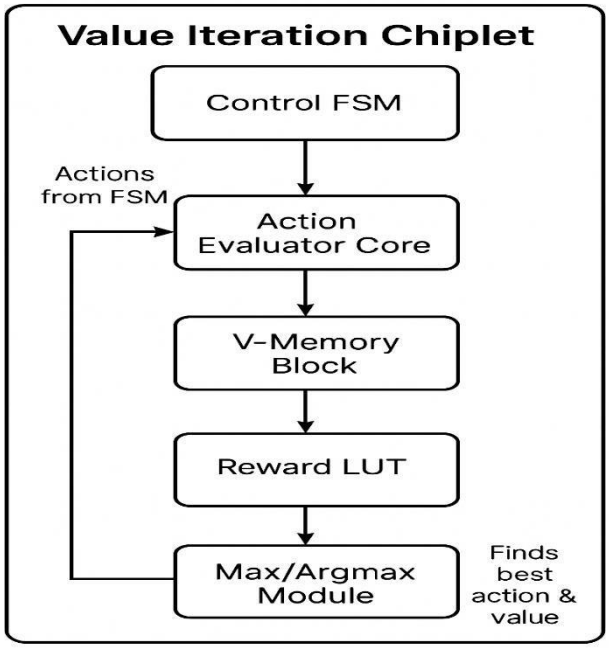
---

2. Verilog Testbench: `tb_ValueIterationCore.v`

This initializes a state with dummy values and checks output.

```
verilog CopyEdit
module tb_ValueIterationCore;

    logic clk = 0, rst = 0, start = 1;
    logic [7:0] gamma = 8'd230; // γ = 0.9 in fixed point
     logic [15:0]
V_in_next[4];     logic [7:0]
R_in[4];     logic [15:0]
V_out;     logic [1:0]
policy_out;     logic done;

    ValueIterationCore dut (
        .clk(clk), .rst(rst), .start(start), .done(done),
        .gamma(gamma),
        .V_in_next(V_in_next),
        .R_in(R_in),
        .V_out(V_out),
        .policy_out(policy_out)
    );
    always #5 clk = ~clk;

    initial begin
        // Simulate a state with 4 possible actions
        V_in_next[0] = 16'd100;  // Q = 0 + γ·100 = 90
        V_in_next[1] = 16'd200;  // Q = 0 + γ·200 = 180
        V_in_next[2] = 16'd150;  // Q = 0 + γ·150 = 135
        V_in_next[3] = 16'd50;   // Q = 0 + γ·50  = 45
        R_in[0] = 0; R_in[1] = 0; R_in[2] = 0; R_in[3] = 0;

        #20;

        $display("Max V[s]: %d, Best Action: %d", V_out, policy_out);
if (V_out != 16'd180 || policy_out != 2'd1)
            $display(" Test Failed");
else
            $display(" Test Passed");

        $finish;
end
endmodule
```

3. Block Diagram Generation



Q-Learning for Frozen Lake Problem

```
import numpy as np import

random import

matplotlib.pyplot as plt


BOARD_ROWS = 5

BOARD_COLS = 5

START = (0, 0)

WIN_STATE = (4, 4)

HOLE_STATE = [(2,0),(4,1),(2,2),(3,3)]


class State:     def __init__(self,

state=START):

      self.state = state

self.isEnd = False
```

```
    def getReward(self):
        for i in HOLE_STATE:
if self.state == i:
                return -5        if
self.state == WIN_STATE:
            return 1
else:
            return -1


    def isEndFunc(self):        if
(self.state == WIN_STATE):
            self.isEnd = True
for i in HOLE_STATE:
if self.state == i:
self.isEnd = True


    def nxtPosition(self, action):
if action == 0:
            nxtState = (self.state[0] - 1, self.state[1]) # up
elif action == 1:
            nxtState = (self.state[0] + 1, self.state[1]) # down
elif action == 2:
            nxtState = (self.state[0], self.state[1] - 1) # left
else:
            nxtState = (self.state[0], self.state[1] + 1) # right


        if (nxtState[0] >= 0) and (nxtState[0] <= 4):
if (nxtState[1] >= 0) and (nxtState[1] <= 4):
```

```
            return nxtState
return self.state  class Agent:
def __init__(self):        self.states
= []        self.actions = [0,1,2,3]
self.State = State()        self.alpha
= 0.5        self.gamma = 0.9
self.epsilon = 0.1        self.isEnd =
self.State.isEnd
self.plot_reward = []        self.Q =
{}        self.new_Q = {}
self.rewards = 0


      for i in range(BOARD_ROWS):
for j in range(BOARD_COLS):                for
k in range(len(self.actions)):
self.Q[(i, j, k)] = 0
self.new_Q[(i, j, k)] = 0        print("Initial
Q-values:")        print(self.Q)


   def Action(self):        rnd =
random.random()
mx_nxt_reward = -10
action = None        if(rnd >
self.epsilon):
          for k in self.actions:                i,j =
self.State.state              nxt_reward =
self.Q[(i,j,k)]              if nxt_reward >=
mx_nxt_reward:
                action = k
```

```
                mx_nxt_reward = nxt_reward
else:
          action = np.random.choice(self.actions)

      position = self.State.nxtPosition(action)
return position, action

   def Q_Learning(self, episodes):
      x = 0        while x <
episodes:           if
self.isEnd:
             reward = self.State.getReward()
self.rewards += reward
self.plot_reward.append(self.rewards)
i,j = self.State.state               for a in self.actions:
self.new_Q[(i,j,a)] = round(reward, 3)
self.State = State()              self.isEnd =
self.State.isEnd               self.rewards = 0
x += 1           else:
             mx_nxt_value = -10
next_state, action = self.Action()
i,j = self.State.state               reward =
self.State.getReward()
self.rewards += reward               for a in
self.actions:
                nxtStateAction = (next_state[0], next_state[1], a)                q_value = (1 -
self.alpha) * self.Q[(i,j,action)] + self.alpha * (reward + self.gamma * self.Q[nxtStateAction])
if q_value >= mx_nxt_value:                    mx_nxt_value = q_value               self.State =
State(state=next_state)              self.State.isEndFunc()              self.isEnd = self.State.isEnd
```

```
self.new_Q[(i,j,action)] = round(mx_nxt_value, 3)            self.Q = self.new_Q.copy()
print("Final Q-values:")        print(self.Q)


   def plot(self, episodes):
plt.plot(self.plot_reward)        plt.title("Cumulative
Reward vs Episodes")        plt.xlabel("Episodes")
plt.ylabel("Cumulative Reward")
       plt.grid()
plt.show()


   def showValues(self):        for i in
range(BOARD_ROWS):
          print('-----------------------------------------------')
out = '| '
          for j in range(BOARD_COLS):
             mx_nxt_value = -10
for a in self.actions:
                nxt_value = self.Q[(i,j,a)]                  if
nxt_value >= mx_nxt_value:
mx_nxt_value = nxt_value                out +=
str(mx_nxt_value).ljust(6) + ' | '            print(out)
print('-----------------------------------------------')
```

```
Final Q-values:

{(0, 0, 0): -5.735, (0, 0, 1): -5.262, (0, 0, 2): -5.735, (0, 0, 3): -5.262, (0, 1, 0): -5.262, (0, 1, 1): -4.736, (0, 1, 2)
-4.736, (0, 2, 0): -4.736, (0, 2, 1): -4.152, (0, 2, 2): -5.262, (0, 2, 3): -4.152, (0, 3, 0): -4.152, (0, 3, 1): -3.503, (0
3, 3): -3.504, (0, 4, 0): -3.503, (0, 4, 1): -2.783, (0, 4, 2): -3.994, (0, 4, 3): -3.484, (1, 0, 0): -5.729, (1, 0, 1): -5.
-5.259, (1, 0, 3): -4.736, (1, 1, 0): -5.262, (1, 1, 1): -4.152, (1, 1, 2): -5.262, (1, 1, 3): -4.152, (1, 2, 0): -4.736, (1
2, 2): -4.736, (1, 2, 3): -3.503, (1, 3, 0): -4.152, (1, 3, 1): -2.782, (1, 3, 2): -4.152, (1, 3, 3): -2.783, (1, 4, 0): -3.
-1.982, (1, 4, 2): -3.503, (1, 4, 3): -2.783, (2, 0, 0): -5, (2, 0, 1): -5, (2, 0, 2): -5, (2, 0, 3): -5, (2, 1, 0): -3.675,
(2, 1, 2): -5.156, (2, 1, 3): -4.25, (2, 2, 0): -5, (2, 2, 1): -5, (2, 2, 2): -5, (2, 2, 3): -5, (2, 3, 0): -3.503, (2, 3, 1
2): -5.499, (2, 3, 3): -1.981, (2, 4, 0): -2.783, (2, 4, 1): -1.091, (2, 4, 2): -2.782, (2, 4, 3): -1.981, (3, 0, 0): -2.75,
(3, 0, 2): -2.778, (3, 0, 3): -2.775, (3, 1, 0): -3.539, (3, 1, 1): -3.0, (3, 1, 2): -3.027, (3, 1, 3): -2.782, (3, 2, 0): -
-1.98, (3, 2, 2): -2.609, (3, 2, 3): -2.75, (3, 3, 0): -5, (3, 3, 1): -5, (3, 3, 2): -5, (3, 3, 3): -5, (3, 4, 0): -1.981, (
4, 2): -5.499, (3, 4, 3): -1.091, (4, 0, 0): -2.829, (4, 0, 1): -3.017, (4, 0, 2): -3.017, (4, 0, 3): -4.125, (4, 1, 0): -5,
1, 2): -5, (4, 1, 3): -5, (4, 2, 0): -2.075, (4, 2, 1): -1.703, (4, 2, 2): -2.75, (4, 2, 3): -1.089, (4, 3, 0): -2.75, (4, 3
2): -0.838, (4, 3, 3): -0.099, (4, 4, 0): 1, (4, 4, 1): 1, (4, 4, 2): 1, (4, 4, 3): 1}
```

```
if __name__ == "__main__":

ag = Agent()    episodes =

10000

ag.Q_Learning(episodes)

ag.plot(episodes)

ag.showValues()
```

```
Run

Initial Q-values:

{(0, 0, 0): 0, (0, 0, 1): 0, (0, 0, 2): 0, (0, 0, 3): 0, (0, 1, 0): 0, (0, 1, 1): 0, (0, 1, 2): 0, (0, 1, 3): 0, (0, 2, 0
2, 2): 0, (0, 2, 3): 0, (0, 3, 0): 0, (0, 3, 1): 0, (0, 3, 2): 0, (0, 3, 3): 0, (0, 4, 0): 0, (0, 4, 1): 0, (0, 4, 2): 0,
0): 0, (1, 0, 1): 0, (1, 0, 2): 0, (1, 0, 3): 0, (1, 1, 0): 0, (1, 1, 1): 0, (1, 1, 2): 0, (1, 1, 3): 0, (1, 2, 0): 0, (1
0, (1, 2, 3): 0, (1, 3, 0): 0, (1, 3, 1): 0, (1, 3, 2): 0, (1, 3, 3): 0, (1, 4, 0): 0, (1, 4, 1): 0, (1, 4, 2): 0, (1, 4,
(2, 0, 1): 0, (2, 0, 2): 0, (2, 0, 3): 0, (2, 1, 0): 0, (2, 1, 1): 0, (2, 1, 2): 0, (2, 1, 3): 0, (2, 2, 0): 0, (2, 2, 1
2, 3): 0, (2, 3, 0): 0, (2, 3, 1): 0, (2, 3, 2): 0, (2, 3, 3): 0, (2, 4, 0): 0, (2, 4, 1): 0, (2, 4, 2): 0, (2, 4, 3): 0,
1): 0, (3, 0, 2): 0, (3, 0, 3): 0, (3, 1, 0): 0, (3, 1, 1): 0, (3, 1, 2): 0, (3, 1, 3): 0, (3, 2, 0): 0, (3, 2, 1): 0, (3
0, (3, 3, 0): 0, (3, 3, 1): 0, (3, 3, 2): 0, (3, 3, 3): 0, (3, 4, 0): 0, (3, 4, 1): 0, (3, 4, 2): 0, (3, 4, 3): 0, (4, 0,
(4, 0, 2): 0, (4, 0, 3): 0, (4, 1, 0): 0, (4, 1, 1): 0, (4, 1, 2): 0, (4, 1, 3): 0, (4, 2, 0): 0, (4, 2, 1): 0, (4, 2, 2,
3, 0): 0, (4, 3, 1): 0, (4, 3, 2): 0, (4, 3, 3): 0, (4, 4, 0): 0, (4, 4, 1): 0, (4, 4, 2): 0, (4, 4, 3): 0}
```

```
-----------------------------------------------
| -5.262 | -4.736 | -4.152 | -3.503 | -2.783 |
-----------------------------------------------
| -4.736 | -4.152 | -3.503 | -2.782 | -1.982 |
-----------------------------------------------
| -5     | -3.503 | -5     | -1.981 | -1.091 |
-----------------------------------------------
| -2.75  | -2.782 | -1.98  | -5     | -0.1   |
-----------------------------------------------
| -2.829 | -5     | -1.089 | -0.099 | 1      |
-----------------------------------------------
```