AI Project1: Development of a superb AI Program for Reversed Reversi

叶璨铭，12011404@mail.sustech.edu.cn

1 Introduction

In this project, we tries to develop a **high-level *Reversed Reversi* program** that are compatible to beat our classmates' programs. **Our purpose is not seeking to prevail over others, but to master the essence of AI system development by learning from and communicating with each other.**

As Kronrod says, computer board game is the "fruit fly in Artificial Intelligence".[1][2] This interesting computer game project will definitely provide me with **a better understanding** of the **knowledge I learnt in the AI course** via my **hand-by-hand practice and experiments** in the procedure of accomplishing this project.

1.1 Problem background

Reversi, also called Othello, is a **deterministic, turn-taking, two-player, zero-sum** game of perfect information.[1] Reversi is not popular in China before the development of the Internet.[3] Although it may not often appears as a board game, it is indeed popular in the research of computer game because of its relatively small search space.[1][2] Computers have always excelled in Reversi because average human players cannot envision the drastic board change caused by move[4], and because human players dislike to risk taking a seemingly bad but actually best move.[5]

Reversed Reversi, also called the anti mode of Othello, shares the same dynamics of the chessboard enviroment as Reversi, while the objective is the opposite.[5] In the game, each player places a piece of his color on the board, flipping any opponent's pieces that are bracketed along a line. The object of the **Reversi** is to have the **most discs** turned to display your color when the last playable empty square is filled, while **Reversed Reversi** expects the winner to have the **least discs.**[6] A formal definition of the game rules of Reversed Reversi will be presented in Part 2.

1.2 Literature review

Before we start to develop our own algorithms for Reversed Reversi, it is necessary for us to do a literature review on how previous researchers develop programs that play Reversi, Reversed Reversi and other board games.

1.2.1 Normal Reversi program

The first world class Reversi program is *IAGO* developed by Rosenbloom in 1982. This program effectively quantified Reversi maxims into efficient algorithms with adversial search techniques.[4] Later in 1990, 李开复& Mahajan developed another program called *BILL*, which far surpassed the *IAGO* by introducing dynamic evaluation function based on Bayesian learning.[4][2] Although *IAGO* and *BILL* are best computer programs that play Reversi at their times, the top human players were not beaten untill the occurence of the program *Logistello* developed by Buro in 1997.[1] The main new idea of *Logistello* is that it automatically learns from previous plays[3][5] **After that, it is generally acknowledged that humans are no match for computers at Reversi.**[1]

In 1997-2006, Andersson developed a practical Reversi program called *WZebra*. It has big opening books, state-of-the-art search algorithms, pattern-based evaluation scheme that can stably run on Windows 2000 to even today's Windows 11 platforms.[7] While it gains a better performance and stability by applying several techniques of C Programming Language, the basic ideas of *WZebra*, however, are no more than *BILL*'s or *Logistello*'s.

1.2.2 Reversed Reversi program

While it is often the case that reversed board game is easy and boring, such as reversed Chinese Chess, Go and Chess, **Reversed Reversi is worth playing and it is an art to play it well**. [5] According to MacGuire, much of the strategic thinking behind the classic game can also be applied to the reverse game, though sometimes in reverse. [8]

Tothello, a program developed by Pittner, is believed to be the best program in the world playing Reversed Reversi untill 2006. [5]

1.2.3 General board game program

2 Preliminary

In the last part, we have known the background of the problem and found some useful references. Next, we need to **formulate** the problem in formal language to **disambiguate the potential confusion.** [6] With the formal logic system, we can then **derive some basic theorems and corollaries** that any Reversed Reversi game must logically follows. With this knowledge in our minds, it is clear how to design our models and experiments in the next sections.

2.1 Problem formulation and notations

Informally, the Reversed Reversi problem is to build a program playing Reversed Reversi with some kinds of high *Intelligence*.

Formally, this problem can be formulated as a **Task Environment,** which is specified by a tuple (P, E, A, S), where P is the performance measure, E is the enviroment, A is the actuators, and S is the sensors.

Besides problem, we also need to formulate the program. The program for this problem can be formulated as an **Agent Algorithm**, which is specify by a function G, mapping the agents' percept histories to its action.

Now we formulate P, E, A, S and G respectively.

2.1.1 Environment E, Actuators A and Sensors S

Environment E, Actuators A and Sensors S are defined by the following notations:

Notation	Interpretation	Restrictions
n	The chessboard size. Reversed Reversi typically has 4x4, 6x6 and 8x8 modes.	$n >= 4 \wedge n \mod 2 = 0$
t	The round number. **Notice that in our convention, round 0-3 exists and is played by the environment. The two agents begin to play at round 4.**	$t \in \mathbb{N} \wedge t <= n^2$
$i = (x, y)$	The row index and the column index.	$I = \{(x, y) \in \mathbb{N}^2 \vert 0 <= x, y < n\}$ $i \in I$
C_t	The color that moves at round t. There are 3 possible colors, 0, 1 and -1. Two agents controls only 1 or -1.	$C = \{0, 1, -1\}$ $C_t \in C \wedge C_t = 2 \cdot (t \mod 2) - 1$
S_t	The state of chessboard at round t. It is a nxn matrix with color values.	$S = I \to C$ $S_t : S$
$S_t(x, y) = S_t((x, y))$	The color at (x, y) on the chessboard at round t.	$S_t(x, y) \in C$
$ACTIONS(s, c)$	A set of legal indexes given chessboard state and the color.	$ACTIONS(s, c) \in 2^I$
$RESULT(s, c, i)$	The result chessboard after placing index i of color c on chessboard s.	$RESULT : \{(s, c, i) \vert i \in ACTIONS(s, c)\} \to S$
$TERMINAL - TEST(s, c)$	Whether the chessboard s is a terminated state of the game. It is not related to c in this game, but sometimes it does.	$TERMINAL - TEST : (S \times -1, 1) \to \{T, F\}$
UTILITY(s, c)	For terminated states, defines the reward or profit for player c. UTILITY for the same board and different color must be zero-sumed.	$UTILITY(s, c) :$ $(\{s \in S \vert TERMINAL - TEST(s)\} \times \{-1, 1\}) \to \mathbb{R}$ $UTILITY(s, c) + UTILITY(s, -c) = 0$
TO, MO	Time out and memory out for agent.	In unit of seconds and in unit of bytes.

On top of restrictions on something like domains and ranges, we have some definition for these functions.

$$TERMINAL - TEST(s) \iff |ACTIONS(s, c)| = 0 \wedge |ACTIONS(s, -c)| = 0$$
$$UTILITY(s, c) = sign(|\{(i, j) \vert s(i, j) == -c\}| - |\{(i, j) \vert s(i, j) == c\}|)$$

As for ACTIONS and RESULT, it is inconvenient to use first-order language to describe. According to Wang, such knowledge is called **Procedural Knowledge**, to be distinguished from Declarative Knowledge, and **it is better to be represented in well-defined and standardized procedural programming language**, such as Python. [9]

Therefore, our formal definitions for this two function are in Python, and for paper shortage you shall read it on my github (look at `actions` and `updated_chessboard` method at the beginning of the file for ACTIONS and RESULT).

> Interlude: 概念辨析——任务环境(Task Environment)和智能体（Agent）的概念的区别与联系
>
> **为什么任务环境和智能体都有感受器（Sensors）和效应器（Actuators）？**
>
> - 你可能注意到了，我们认为感受器和效应器是任务环境的一部分，而感受器和效应器也是Agent的一部分。[1] 《人工智能现代方法》中这个令人困惑的表述是怎么回事呢?
> - 其实很容易理解。
> - 任务环境是一个对Agent要解决的问题的一个模型，为Agent解决问题提供了一个平台。

  - 在本次Project中，我们可以认为老师和学助提供的OJ对战平台和最后用同学之间两两对战的方式给我们Project打分的方式就是一个完整的任务环境。
  - 我们刚才做的就是对这个任务环境的形式化描述。
  - 任务环境的风格直接影响到Agent程序的适当设计.[1]
    - 因此，OJ对战平台使用什么接口调用我们的Agent程序、提供了什么信息、有什么规则是已知的，这些任务环境的性质都影响着我们的编程。
- 智能体包含传感器、效应器和算法。每一时刻Agent接收感知之后通过算法将感知序列映射到其效应器应采取的行动。[1]
  - Agent的感受器和效应器包含在Task Environment当中，确实是两个有交叉的概念。[1] **因此我们的形式化定义当中只把Agent的Algorithm G单独拿出来和Task Environment并列。**
  - AIMA一书选择这么区分，在于Agent的理性程度的度量、Agent的设计方法，都与任务环境中对Agent的传感器与效应器的限制有关。[1]
- **因此，可以认为OJ对战平台允许我们通过改写`go(chessboard)->candidate_list`的方法来获得棋盘信息、返回控制信息，就是任务环境规定的Agent必须拥有的感受器和效应器.**
  - 此外，尽管并不是很明显，**OJ服务器的内存的读写、CPU的使用，也是我们Agent的传感器和效应器。**作为任务环境的一部分，它限制我们的Agent必须在合理时间内(5s)以合理的存储(100MB)计算得到结果。

2.1.2 Agent Algorithm G

G is a function that maps percept histories to an action. We will implement this function by search model.

$$
\begin{gathered}
A = P^t \to I \\
G : A \\
P = S \times \{1, -1\}
\end{gathered}
$$

2.1.3 Performance P

There are 4 ways to define intelligence, and we choose rationality of actioning. [1] And to measure the rationality of an agent, we needs not only performance, but also the knowlege, action, and perception that an agent is able to obtain. [1] Approximately, performance is the most important measure for rationality.

In this project, 30 agent algorithms, numbered $G_0, G_1, \ldots, G_{29}$ from 30 students are uploaded to the OJ, and 30 agents are played in round robin, i.e. each agents are obliged to playe with another agent exactly twice. [6]

$$
\begin{gathered}
SCORE(G_0, G_1) = (UTILITY(s_0, c_0) + UTILITY(s_0, -c_0), \\
UTILITY(s_1, c_1) + UTILITY(s_0, -c_1), 0, \ldots, 0) \\
SCORES(G_{0:29}) = \sum_{0 \le i < j \le 29} SCORE(G_i, G_j)
\end{gathered}
$$

where s_0 is the final chessboard when G_0 is of color c_0, and s_1 is the final chessobard when G_0 is of color c_1.

So the performance for our agent, say G_i is

$$
Perf(G_i) = SCORES(G_{0:29})[i] / sum(SCORES(G_{0:29}))
$$

Interlude: 概念辨析——任务环境(Task Environment)和智能体（Agent）的概念的区别与联系(续)

任务环境的Performance和Agent内部的Performance element的区别与联系？

在2.4.6 AIMA提到Learning Agent的内部也有Performance模块。[1]

- 请注意，Learning Agent内部的Performance模块往往与任务环境中我们所说的Performance P不同。
- 以本次Project为例，任务环境的Performance是指截止日期到时，我们的程序与同学程序两两对打胜利的次数。然而这个Performance的衡量在平时我们改进自己程序的时候是不可用的（或者计算代价很大）。
- 这样的Performance是问题本身（任务环境）的要求，是我们的目标，但是不具备可计算性。
- 而我们为了让Performance具有可计算性，在本地提出**性能模型（Performance Model）**，通过简化假设让Performance的计算具有可能，从而改进我们自己的程序，这就是Learning Agent的Performance element.
- 如果我们的模型假设合理，模型建立准确，就可以达到本地改进后实际上也能在OJ上拿到更多分数的目的。本次Project我提出了三种有效的性能模型，请见3.3.2 。

2.1.4 Nature of the task environment

- **Fully obersvable.** Agents are able to know the exact information of the chessboard, which is the almost the only state in the environment. However, according to the minimax theorem, if the agent algorithms of others are known to our agent, our agents would be possible to have a better minimax search strategy. [9] [1] In this sense, the environment is only partially observable.
- **Deterministic.** The state of the enviroment is totally determined by the agents that are fighting.
- **Multi-Agent**. In every fighting, it is an 2-agent. As for the performance, it is 30-agent.
- **Episodic and Sequential.** Matches are independent to each other, so episodic. Moves in one match are dependent, so sequential.
- **Semi-dynamic**. The chessboard is static, but their is a TO for our agent., which means the counter of time as a state of the environment is dynamically changing.
- **Discrete.**The states of all chessboard are finite and discrete.
- **Known.**The rule of the game is known by the agent, in the sense that the agent models the ACTIONS and RESULT funcions accurately.

2.2 Basic theorems and corollaries

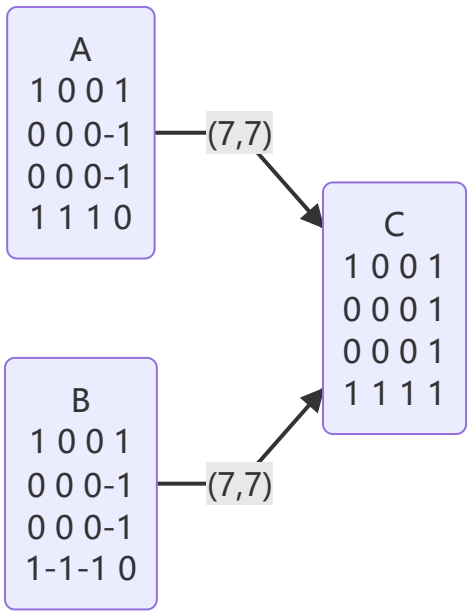
2.2.1 Environment Theorems

Theorem 1. *Monotonicity of the number of total chess pieces.* The total chess piece number (count if non zero places of the S_t)is a function of round t, and as round goes up, it monotonically increases. This is because the RESULT function only places a new piece or convert the old pieces into another color.

Corollary 1. *Rounds are always total pieces counts.* At any round t, we have $t = \sum_i |S_t(i)|$.

Theorem 2. *RESULT function is not invertible.* Precise retract from current chessboard is impossible if you don' t have the information of previous chessboards or previous moves.

Proof 2. Proof by counter example. Let n=8. We show the **right down** corner of 4x4 chessboard as follows



We can see that configuration A and configuration B are different, but after one move, they both became configuration C.

In addition, we should show that configuration A and B are possible states to make this proof valid. This can be easily done by writing a search program, for the sake of space limit, we don't go through it.

2.2.2 Performance Theorems

Definition. *Beating relation.* Beating relation is a relation over the set of all agent algorithms. It is satified for algorithm G1 and G2 IFF G1's accumulated UTILITY is greater or equal to G2's in competitions that takes black color in turn. If G1 and G2 are not deterministic algorithms, then we calculate the expected accummulated UTILITY.

Theorem 1. *Rock Paper Scissors.* The Beating relation B over the set of all agent algorithms is not a transitive relation, thus not a partial order on the algorithm set.

2.2.3 Agent Algorithm Theorems

Definition 1. *Minimax Estimate Function* $m\hat{v}_{d,e}$ is defined recursively as following:

1. Basic step.

$$\forall e \subseteq S \times R, \forall c \in C,$$
$$m\hat{v}_{0,e}(s,c) = e(s,c).$$

   The zero-depth minimax estimate is just the value of evaluation function for the node.

2. Induction.

$$\forall e \subseteq S \times R, \forall d \in Depth, \forall c \in C,$$
$$m\hat{v_{d+1,e}}(s,c) = max_{x \in RESULT(s,c,ACTIONS(s))}(-\hat{mv}_{d,e}(x,-c))$$

You may notice that our mimimax definition always uses max instead of max for MAX and min for MIN as AIMA book do.[1] Our version of definition, called maxi-one's-own, is equivalent and better in the following aspects:

- Minimax estimate function is no more than **a special evaluation function, except that it needs recursion and RESULT of chessboards to obtain its value**. Since evaluation function values differ as color differs, it make no sense for minimax estimate function to be without color and relys on the environment given color.

- In the perspective of Software Engineering, it is **a bad practice to write hundreds of duplicate and volatile code with just sign changing**, because it makes testing harder and introduces bugs of 2 function inconsistency.
- In the view of **Numba**, a single recursive function is **much easier to compile** than 2 recursive funcitons calling each other. [10]
- Sometimes games involve **more than 2 players.** In this case, each person haves their own values on the same chessboard, and they both wants to maximize their value, so the game tree is drawn with a floor order 0, 1,2 and each node decides according to its color. In the case of exactly 2 player, their values must be opposite, so minimax makes sense, [1]

3. Bound check.

$$If\ d > max(Depth), define\ m\hat{v}_{d,e}(s,c) = mv\hat{}_{d-1,e}(s,c)$$

Theorem 1. *Deeper search doesn't imply better performance.*

Proof 1. *Prove by counter example.* Consider the following game tree:

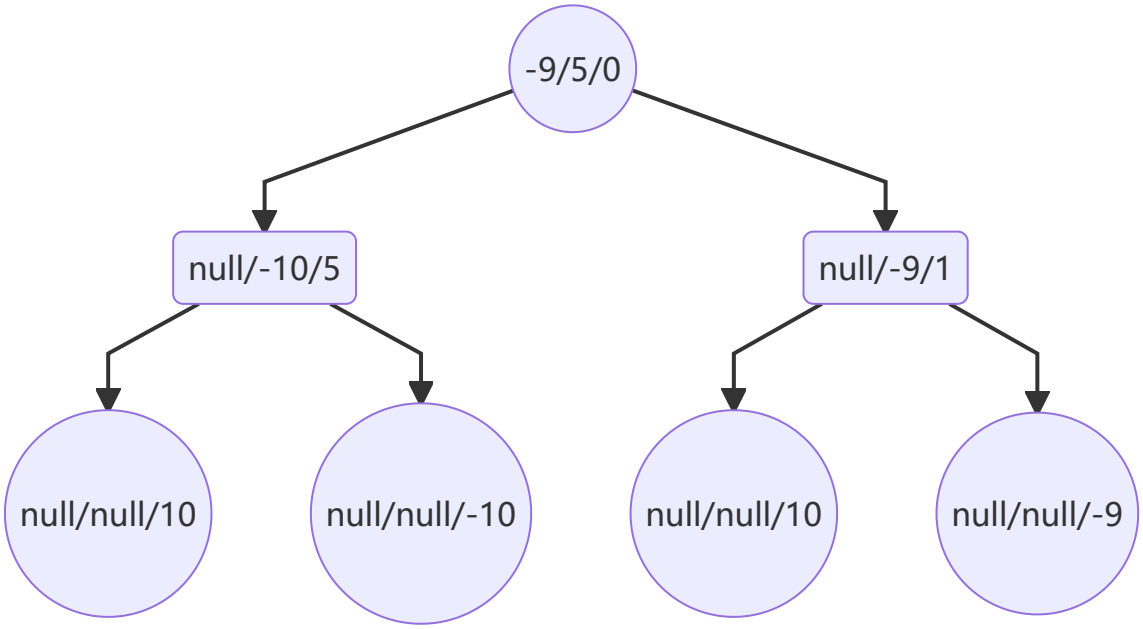


Figure1. In each node s, we display their *minimax estimate function* value of depth 2/1/0. We applied the Definition 1 except that for clarity the rule 3. bound check is not enabled and that the **values are all of color of the root node.** Instead, **depth that are illegal for current node is diplayed as** `null`.

As you can see in the figure1, the node at depth 0, 1 or 2 are able to evaluate their *minimax estimate function* value of depth 2/1/0. If the game ends up at depth 2, the deeper estimation does gain the same estimation orderasf the the shallower ones.

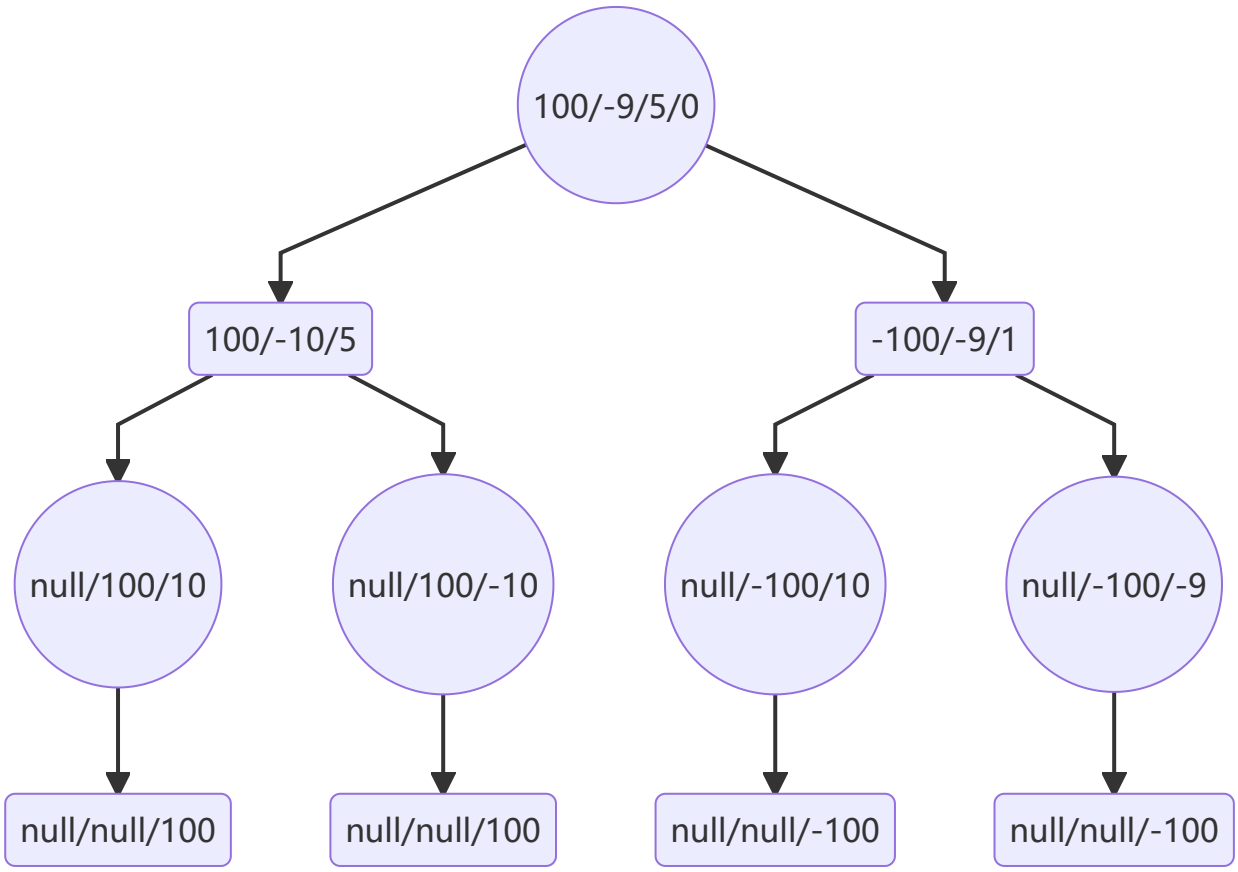


Figure2. The case when the game terminates at the depth 3.

Now we consider the game to end at depth 3. The zero-depth estimation of nodes that are of level 3 indicates that the root node should choose the left child to win. A three-depth and a one-depth estimation of the root node gives the correct result. The two-depth estimation of the root node, however, gives the wrong answer.

Therefore, the two-depth estimation performs poorer than the one-depth estimation in this case, which implies that deeper search doesn't imply better performance.

Definition 2. *The Accurate or True Minimax Function* is defined as

$$mv(s,c) = \hat{mv}_{n^2,UTILITY}(s,c)$$

Theorem 2. *Minimax estimates itself.* Minimax function is a fixed-point of the Minimax estimate function on the set of evaluation functions.

$$\forall s \in S, \forall d \in N, \forall c \in C, mv(s,c) = \hat{mv}_{d,mv}(s,c)$$

Theorem 2 can be proved by mathematical induction.

3 Methodology

3.1 General work flow

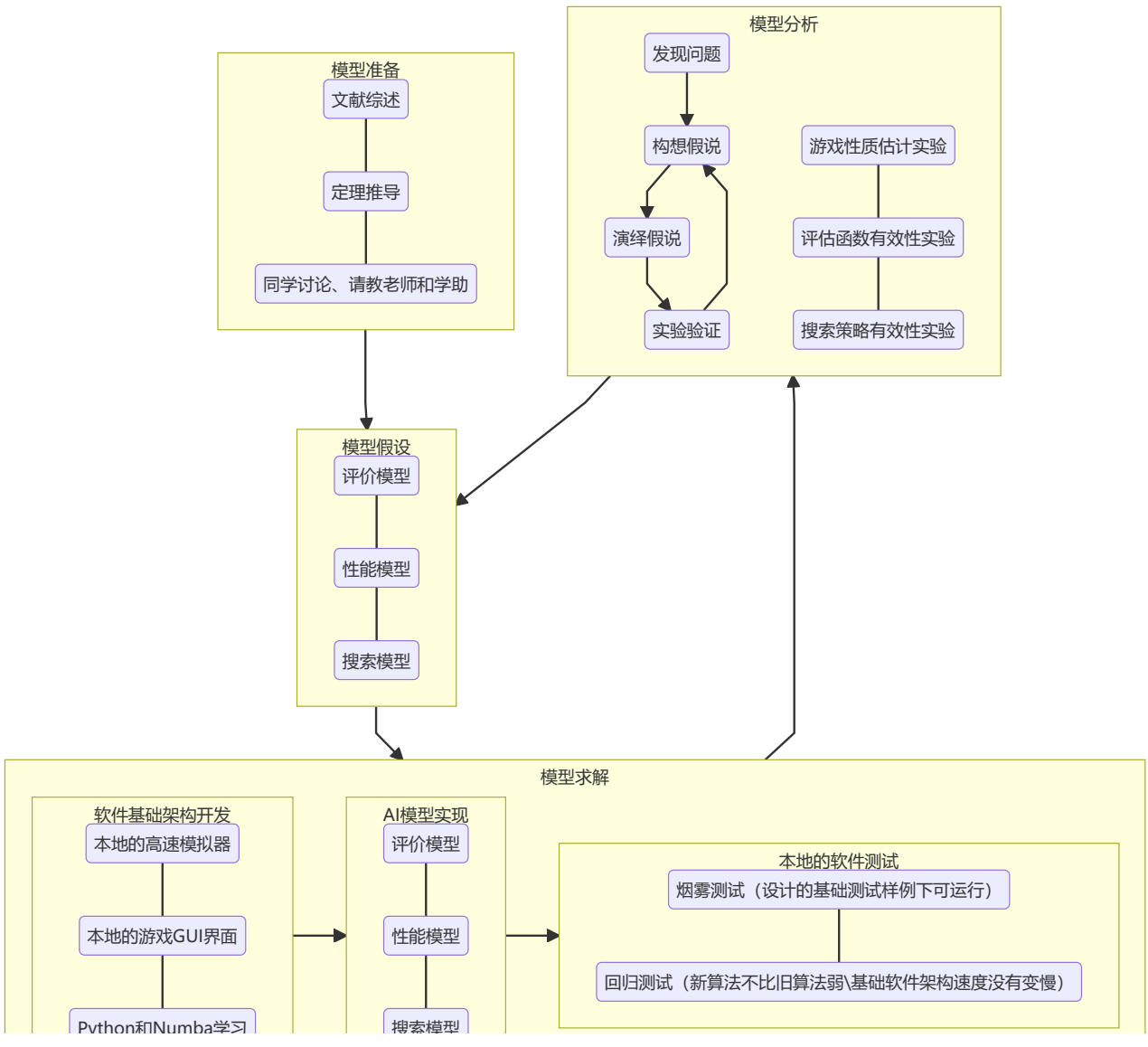
After having a literature review and doing a formulation, I find these problems to be vital in this project:

- Online

- How to design an efficient and effective algorithm to **search adversarially** on the game tree? How to write it in Python with no bugs?
- How to design a good **evaluation** function?

- Offline
  - How to **measure** the intelligence of a Reversed Reversi program?
  - How to utilize **local search** algorithms to find the best weight in evaluation function?
  - How to generate opening books and the weights for pattern-based evaluation scheme by inverse **adversarial searching**?

Therfore, our genearl workflow are as follows:



3.2 Hypothesis and assumptions

Hypothesis 1. *Markov Assumption.* If we models the agent algorithm as a policy function $\pi(s,a) = Pr\{a_t = a|s_t = s\}$, Markov assumption on this function will assume that $Pr\{a_t = a|s_{1:t} = s\} = Pr\{a_t = a|s_t = s\}$ .Notice that we don't need the complete Markov Decision Process because the enviroment is after all deterministic and fully observable, and we just assumes the decision probability is irrlevant of previous state history.

Assumption 1. *Evaluation function is on the current state.* In other words, we don't consider evaluation on actions itself, nor do we consider evaluation on percept histories.

> Note: Hypothesis 1 and Assumption 1 are **not obvious and not theorems!** From section 2.2 Environment Theorems 2, we know that the RESULT function is not invertible, so when we decide our action, the actioned chessboard is lack of information of the previous step.
>
> We make this assumption because
>
> - From the perspective of Minimax, the Minimax value for all chessboard states are determined, **we estimate it with uncertainty because our lack of computational resource.** [1] **If we were able to have computational resource, we surely won't consider action and percept history.** Therefore it is reasonable to assume that action is helpless to the estimation of minimax value.
> - Most papers do so on Reversi.
> - This assumption makes the evaluation model simpler.

Hypothesis 2. *The deeper, the better.* We assume a good evaluation function doesn't go worse when it reaches a deeper search.

Hypothesis 3. *Don't give up mobility.* Mobility is important and more mobility means better game in Reversi. Then most of my classmates thinks that giving up the mobility means a better game in Reversed Reversi. That opinion is wrong.

MacGuire points out that if you can achieve a position where you can restrict the availability of moves to your opponent then you are well on the way to victory. [8] What he mean by saying this is that you should restrict the mobility of your opponent, in order to force the opponent to place his move at a place he doesn't want, for example the corner.

Lee and Mahajan has a similar idea that the mobility must have a weight to indicate its importance. [4] Therefore, the key change for normal to reversed is not from more mobility to less mobility, but from weight of mobility importance of Reversi to weight of mobility importance of Reversed Reversi .

3.3 Model design

3.3.1 Evaluation Model

The evaluation function must conform to three rules, according to AIMA. [1]

- The evaluation function must give the same order as the Accurate Minimax Function **at least for the terminated states.**
- The evaluation function must not compute for a long time.
- The evaluation function must be related to the win rate.

I partially agree with AIMA's opinions. If the

3.3.2 Performance Models

Here four models that measure the performance of the evaluation model are presented. Among them, I proposed model 1, 2, and 3, and model 4 was found in a paper. Since I haven't learnt machine learning in class yet, I only used model 1 and 2 for experiments.

3.3.2.1 Performance Model 1: Round Robin First Color in Turn Measure

A good performance model must at least gives a partial order of what it profiles on. **From Theorem 1. in section 2.2.2, we know that the "beaten relation" is a pretty bad performance model.** Then, how can we measure the performance reliably?

A simple idea that most of my classmates and I could figure out was round robin. Two

3.3.2.2 Performance Model 2: Improved Monte Carlo Win Rate Estimation Measure

3.3.2.3 Performance Model 3: Structural Minimax Value Loss Measure

We know from Theorem 2 of section 2.2.3 that Accurate Minimax Function is the fixed point of Minimax Estimate Function of any depth. Therefore, the best evaluation function is the Minimax Function.

Machine learning learns functions from the data. And the Minimax Function is no more than a normal evaluation function that takes chessboard state and color as the inputs and outputs a value, except that its primary definition is recursive. So if we are able to design a Machine Learning Model that fits Minimax Function well, it must be exicting.

One of the cruxes of the problems is the dataset. We can generate the Minimax value from the last turn of the game and goes back to at most 12 plies if we have 5 seconds; if we had more time, more plies of data would be genearated. However, the data that we don't generate, such as a chessboard at a very beginning round number, will not be learnt by the agent, and thus agent would perform bad in those rounds. After all, even if the model is perfect, data, or the time we have to generate data, limits the performance of the learning agent.

Another problem is that, how do we define the loss function, i.e. Performance Model? I think the difference of the labeled value and the predicted value is not enough. Instead, we need a Structural Minimax Value Loss Measure that utilize the fixed point nature of the Minimax Function. Structural Minimax Value Loss Measure is computed by

1. The proposed evaluation function estimates n plies of the nodes on a game tree rooted at state.
2. For every node, there is a true minimax value in our dataset. We compute the loss for every nodes.
3. We accumulates the loss of all nodes and credit it to the root state.

3.3.2.4 Performance Model 4: Search Depth Measure

As we know, a better evaluation function implies a better pruning for alpha-beta search. [1] So it make sense that **when the search depth goes deeper after using a new evaluation function with the same search method(with alpha-beta search and move sorting), the new evaluation function is a better estimate of Minimax value**.

Actually, this is one of the performance models used by BILL. Lee & Mahajan argues that **Bayesian learning is instrumental to BILL's playing strength because results showed that it improved BILL's play by two plies of search.** [4]

3.3.3 Search Model

3.3.3.1 Iterative Deepening Search For Timing

```
function Double-Alpha-Maximax-Search(state, color, time_out, q) return a value
and a move
    inputs: state in S, color in C: our value is computed on a
                chessboard state in the view of color.
            time_out in R, maximum allowed time in unit of seconds.
            q in R, a parameter used to balance time and search.
    current_depth = 2
    time_out *= q
    try
        do
            last_move = move
            value, move = Double-Alpha-Maximax-Search(state, color,
current_depth, (-1, -1))
            current_depth +=1
            assumed_breadth = average_breadth(rounds, current_depth) // 宽度表明了
多搜一层需要的时间倍数上界。
        while(time_used*assumed_breadth < q*time_out or (last_move != move)) and
(
                time_used /q < time_out) and (value!=1)
    catch Timeout
        do nothing
    end try
    return value, move
```

3.3.3.2 Double-Alpha Maximax Search

This is an implementation of *Minimax Estimate Function* $m\hat{v}_{d,e}$ defined in section 2.2.3, with **alpha-beta-pruning**, **move sorting**, and **table lookup**. In addition, we designed the UTILITY value of terminal state in order to integrate an **end-game optimal search** into the model.

```
function Double-Alpha-Maximax-Search(state, color, remaining_depth, alphas)
return a value and a move
    persist: evaluation in E, a function that maps state and color to value.
             table, a hash table that maps chessboard to value calculated in the
last step of iterative deepening search.
```

```
    inputs: state in S, color in C: our value is computed on a
                chessboard state in the view of color.
            remaining_depth in N, for convenience, we require remaining_depth
is at least 1.
            alphas in [-1,1]^2, a tuple with 2 elements. The 0th is the max
available               value that color -1 can have, The 1th is the max
available value                 that color 1 can have.

    if TERMINAL-TEST(state)then
       return MORE-SIGNIFICANT(UTILITY(state, color)), null  // Once terminal
state is found, the value should be more-significant than evaluation. In our
case, UTILITY is either 0, 1, or -1, and evaluation must be in [-1, 1], so MORE-
SIGNIFICANT is just Identity.
    end if
    acts = ACTION(state, color)
    if EMPTY?(acts) then
        v, m = Double-Alpha-Maximax-Search(state, -color, remaining_depth-1,
alphas)
        v = -v
        if len(table)<M then
            table[state] = v*color
        end if
        return v, null
    end if
    new_chessboards = RESULT(state, color, ACTIONS(state, color))
    sort acts and new_chessboards together,
        with the value of each new_chessboard in table first then value in
evaluation. // For example,  let key= table[state] * current_color + 2 if v is
not null else evaluation(state, current_color)
    if remaining_depth ==1 then return evaluation(new_chessboards[0], color),
acts[0]
    end if
    v, m = lb, null // evaluation function is only allow to in [-1,1], so lb=-1.
Well, -inf is also OK, but is not necessary as the domains are well defined.
    self, other = (color+1)//2, (-color+1)//2
    for action, new_chessboard together in acts and new_chessboards
        nv, t = Double-Alpha-Maximax-Search(state, -color, remaining_depth-1,
alphas)
        nv = -nv
        if len(table)<M then
            table[state] = v*color
        end if
        if nv > v then
            v, m = nv, action
            alphas[self] = max(alphas[self], v)
        end if
```

```
        //When another node along the path with the other color accepts at least
alphas[other] in his point of view, our lower bound v is interpreted as an upper
bound -v that may be rejected by that node. This is alpha-beta pruning, but in a
more generalized form.
        if -v<=alphas[other] then
            return v, m
        end if
    end for
```

3.4 Model analysis

4 Experiments

The following experiments were performed under the following environment conditions：

- Systeminfo：AMD Ryzen 7 4800H with Radeon Graphics， Microsoft Windows 11 专业版, RAM 16G
- Python：Python 3.10.5, Pycharm 2022.1.3 (PE), numpy 1.22.4, numba 0.55.2

Get a hand by hand reproduction of my experiment by cloning the repository and run codes and notebooks in experiment folder!

4.1 Statistical game nature estimation experiment(基于统计的游戏性质估计实验)

General Steps in the following experiments.

1. Open `experiment/游戏随机性质分析/游戏性质随机分析.ipynb`
2. We built an efficient `Simulator` with account function support in ``
3. Write an account function for a new experiment, and runs `do_account_tenth` to experiment ten times.
4. See the result by `print` ing the variables that your account function accounts.

4.1.1 Estimate the average breadth（估计平均行动力）

4.1.2 Estimate the average size of the game tree of n plies after alpha-beta pruning（估计Alpha Beta剪枝后博弈树节点数量）

4.1.2.1 Experiment principle, experiment hypothesis and experiment purpose.

Principle.

Knuth&Moore proved that, with an ideal move sorting, alpha beta pruning has a time complexity of $O(b^{\frac{m}{2}})$. [1] Research also shows that alpha-beta boosts up the search of Tic-Tac-Tok by 15 times. [2]

Hypothesis. Alpha beta pruning drops down the effective branch breadth to be its square root. [1]

Purpose. The purpose of this experiment was

- To see how much times alpha beta pruning do boost up the Reverse Reversi search
- To verify the hypothesis.
- To guide the design of an iterative deepening algorithm.

4.1.2.2 Experiment steps

1. Write a new account function for nodes.

```
a_b_nodes = np.zeros(65)
a_b_nodes_times = np.zeros(65)
def account_a_b_nodes(rounds, color, chessboard, agents):
    global a_b_nodes, a_b_nodes_times
    a_b_node = np.zeros(1)
    alpha_beta_search_account(chessboard, color, a_b_node, 8)
    a_b_nodes[rounds] += a_b_node[0]
    a_b_nodes_times[rounds]+=1
```

2. Add the new account function into the `accoutants` list in `do_account_tenth`

```
accountants = [account_a_b_nodes]
```

3. Run `do_account_tenth` until the precision requirement is satisfied.

4.1.2.3 Experiment results

The average node numbers of each rounds are

```
array([0.00000000e+00, 0.00000000e+00, 0.00000000e+00, 0.00000000e+00,
       4.71800000e+03, 3.41720000e+03, 5.96620000e+03, 8.07735000e+03,
       1.15568000e+04, 1.08959000e+04, 1.47169000e+04, 2.01351500e+04,
       2.18448000e+04, 2.79829500e+04, 3.17610500e+04, 3.35721000e+04,
       3.46187500e+04, 4.71068500e+04, 3.80531500e+04, 5.27592000e+04,
       4.40333500e+04, 4.62005500e+04, 4.10718000e+04, 4.82063500e+04,
       4.25357000e+04, 5.59500500e+04, 4.48639000e+04, 5.51135000e+04,
       5.21724500e+04, 5.16872500e+04, 4.67959000e+04, 5.52873000e+04,
       4.54007500e+04, 5.08628000e+04, 4.43237000e+04, 5.52107500e+04,
       4.11963500e+04, 4.57738500e+04, 3.34940000e+04, 4.21815000e+04,
       3.37077500e+04, 3.63669500e+04, 2.82919000e+04, 2.80875500e+04,
       2.20766000e+04, 2.25810500e+04, 1.83278500e+04, 1.57265000e+04,
       1.02019500e+04, 1.04575500e+04, 7.50470000e+03, 5.45215000e+03,
       4.55930000e+03, 3.54845000e+03, 2.61700000e+03, 1.93475000e+03,
       1.07585000e+03, 4.36300000e+02, 1.82714286e+02, 7.19047619e+01,
       2.67727273e+01, 1.08000000e+01, 5.08695652e+00, 2.16666667e+00,
       0.00000000e+00])
```

4.1.2.4 Experiment analysis

Deduction. Without pruning, the expected nodes for a 8 plies search is

$$nodes \approx b^d \approx 8.093^8 = 18,402,478.675$$

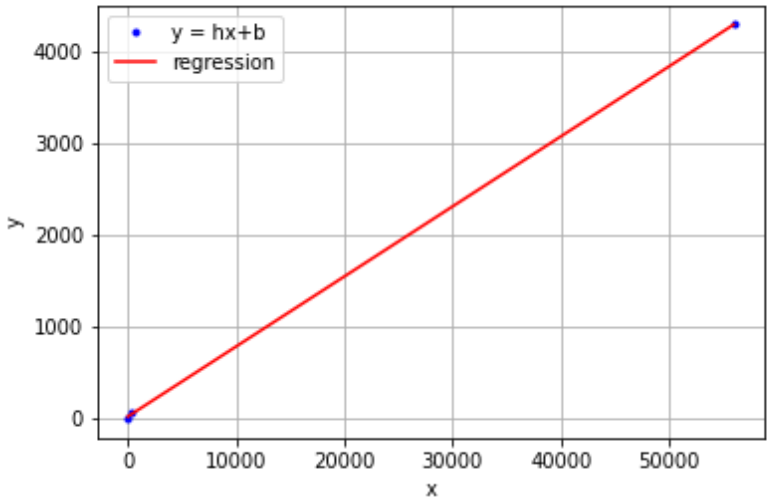
If the hypothesis is correct, the expected nodes for a 8 plies search is

$$a\_b\_nodes \approx \sqrt{b}^d \approx \sqrt{8.093}^8 = 4,289.8110$$

The average node number is here, however, is **55950.05**. So big! Don't panic! This is not because our evaluation function is bad. Rather, this is because the coefficient of the time complexity is unknown to us.

In other words, $nodes = h \cdot \sqrt{b}^{d} + b$. And we shall see if different depth experiments gives us a acceptable(R^2) linear regression for that formula.

depth	hypothesis	actually	times
8	$4,289.8110 \cdot h + b$	55950.05	$55950.05/296.35 = 188.79 \approx 295.35$
4	$65.4966 \cdot h + b$	296.35	$296.35/14.55 = 20.37 \approx 14.55$
2	$1 \cdot h + b$	14.55	



Conclusion.

1. The experiment **supports the hypothesis** that the effective breadth is squarely rooted. The coefficients (h, b) for the time complexity is approximately (0.07629278861048339, 21.33424487500785).
2. Statistics show that the nodes is dropped down by $18402478.675/55950.05 = 328.909$ times. A big improvement!

4.1.3 估计胜者特征：平均占角数与稳定子数

4.2 Explore the influence of different evaluation functions on Agent performance when the algorithm is greedy algorithm (探究当算法为贪心算法时不同评估函数对Agent性能的影响)

4.2.1 Experiment principle

In this part, we do experiments to verify wheter a characteristic is incremental to the performance of our program.

We control the variable by

- 自变量: 评估函数的类型
- 无关变量：搜索算法的类型
- 因变量：三种Performance Model衡量下Agent的表现

4.2.2 Experiment setup and experiment steps

As we said in 4.1.1, one of the best ways to evaluate an RR agents` rationality is to evaluate their ranking in the round robin. We need to

4.2.2.1 Sub-Experiment 1.1 Baseline verification

4.2.2.2 Sub-Experiment 1.2 AHP combination for better

4.2.2.3 Sub-Experiment 1.3 Genetic programming for local search

4.2.3 Experiment results and experiment analysis

4.3 Explore the influence of different search strategies on Agent performance when the evaluation function is constant (探究当评估函数一定时，不同搜索策略对Agent性能的影响)

5 Conclusion and discussion

In conclusion, the three models I built have these advantages and disa

Model	Advantages	Disadvantages
Evaluation model	concise and easy to implement.	too less parameters, too less characteristics compared to state-of-the-art Reversed Reversi Programs.
Performance model	Round robin Model is what OJ Performance uses. Random model is reasonable and by Law of large Numbers.	Round robin is limited for accessible players. Random algorithm is too weak even with a end game search. Random algorithm need a long time to reach a acceptable precision.
Search model	Reasonable timing strategy. Best search depth among my classmates that uses alpha beta search scheme(I reach a average depth of 9 and 12 in the middle game and the end game).	20% of the time is not used. We can probably run MCTS at the rest of the time in further study.

Experimental results match my expectation.

In this project, I have

- gained a better understanding of the knowledge I learnt in the AI course via my hand-by-hand practice and experiments
  - Local search.
  - Adversarial search.
  - and I feel very interested in the upcoming Machine Learning Chapters!
- learnt practical coding skills of Python programming.
  - how to write code that compiles in Numba
  - how much performance can I expect from Numba
  - how slow Python is compared to Java/C++/C.

In further study, I might try to apply new knowledge about machine learning that will be learnt from the next class to implement the idea I talked about in Performance Model 3.

References

1. Stuartj. Russell, PeterNorvig, 诺维格, 罗素, 祝恩和殷建平, 《人工智能:一种现代的方法》, 清华大学出版社, 2013, doi: 9787302331094. ↩

2. 彭之军, 《计算机博弈算法在黑白棋中的应用》, 现代信息科技, 卷 5, 期 17, 页 73-77+81, 2021, doi: 10.19850/j.cnki.2096-4706.2021.17.018. ↩ ↩ ↩ ↩

3. “黑白棋,” 百度百科. https://baike.baidu.com/item/%E9%BB%91%E7%99%BD%E6%A3%8B/80689 (accessed Jul. 16, 2022). ↩ ↩

4. K.-F. Lee and S. Mahajan, “The development of a world class Othello program,” *Artificial Intelligence*, vol. 43, no. 1, pp. 21 – 36, Apr. 1990, doi: 10.1016/0004-3702(90)90068-B. ↩ ↩ ↩ ↩ ↩

5. Pittner. “Reversed othello games.” http://www.tothello.com/html/reversed_othello_games.html (accessed Jul. 16, 2022). ↩ ↩ ↩ ↩ ↩

6. Teachers and assistants in SUSTech. Project-Reversed-Reversi-EN.pdf ↩ ↩ ↩

7. Andersson. “Gunnar’s Othello page.” http://www.radagast.se/othello/ (accessed Jul. 17, 2022). ↩

8. MacGuire. “Strategy Guide for Reversi & Reversed Reversi.” https://www.samsoft.org.uk/reversi/strategy.htm (accessed Jul. 16, 2022). ↩ ↩

9. 学堂在线, “人工智能原理 - 北京大学,” 学堂在线. https://www.xuetangx.com/learn/PKU08091000777/PKU08091000777/10322346/video/17400688 (accessed Jul. 17, 2022). ↩ ↩

10. “8.2.5. NBEP 6: Typing Recursion — Numba 0.38.0rc1+0.ga0390f6.dirty documentation.” http://numba.pydata.org/numba-doc/0.38.0/proposals/typing_recursion.html (accessed Jul. 17, 2022). ↩