Efficient Solver for Dynamic Puzzle Games: Baba Is Y'all

Yuhang Lin, Xianqing Zeng

***Abstract*—The demand for intelligent solvers for puzzle games has overgrown with the increase in depth and complexity of these games. Most traditional puzzles with static mechanisms already have many high-performance algorithms to find solutions. However, puzzles with dynamic rule systems have occurred recently and have only a few algorithms to obtain a solution. Because of the unique dynamic mechanisms, the game *Baba Is You* is chosen as the research object. Two search algorithms, RHEA and MCTS, are performed in this game. MCTS reaches better winning rate than the four baselines, BFS, DFS, Default, and Random. But RHEA performs worst in experiments on a large and overall test-level set. Besides search algorithms, reinforcement learning will be applied in this game in the future.**

***Index Terms*—Baba Is You, Agent, Dynamic Rule, RHEA, MCTS.**

I. Introduction

WITH the increase in depth and complexity of puzzle games, answers to these puzzles require more intelligence and better problem-solving ability. More variables and limitations in consideration make it hard to calculate and obtain correct answers manually and may take more time. Thus the solver agent arouses people's attention. It can help people find solutions quickly and help puzzle designers better understand players' behaviors during the game process, which contributes to improvements to game design and makes it more fascinating and challenging.

A. Puzzle Category

According to the types of rules, puzzles divide into two categories: puzzles with static mechanisms and dynamic mechanisms.

Most puzzle games have a set system of rules that can be followed and will never be changed at any point during play. One of the most classic examples is *Sokoban* because of its discretized, simplistic movement and consistent rule space. In this game, the player can only push rather than pull each crate to designated positions, and this push-pull mechanism will never change by players' actions or states. These straightforward rules staying consistent from level to level are called "static mechanisms".

"Dynamic mechanism" refers to dynamically changing mechanic space affected by the player's actions. These actions may cause temporary or permanent effects on the rule system. Thus rules do not have to be initially made at the start of the game and could be manipulated at any point while the player attempts to solve the level. Some examples of this type of mechanic space include resource changes, gravity changes, and time-sequential changes. One puzzle game with perspective changes is *Monument Valley* as shown in Fig.1 and Fig.2, where the player has to rotate the entire map to navigate through the level successfully. [1]

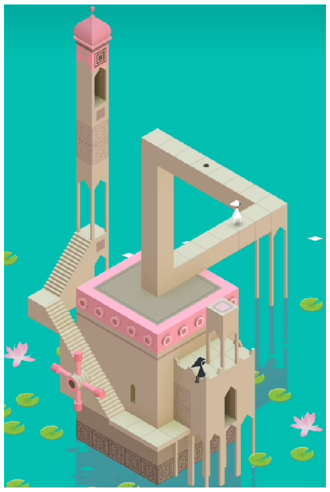


Fig. 1: Before Rotating

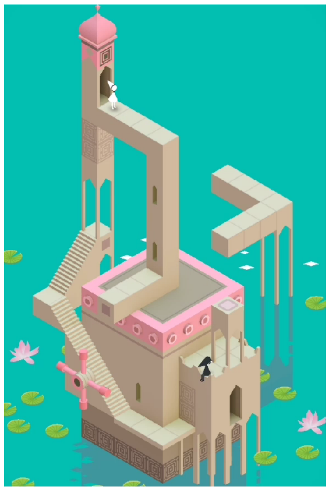


Fig. 2: After Rotating

B. Algorithms for puzzle games

Because of the research and study of static rules over a long history, the solution space of traditional games has been comprehensively explored, and many efficient and high-performance algorithms have already existed. This type of agent consists of the tree search algorithm, reinforcement learning, evolutionary algorithms, etc. [2]

However, since dynamic rule games have appeared in recent years, they have not been studied thoroughly. Some AI competitions exist for adaptive agents, like *Hanabi* (a multi-agent competitive strategy game) [3] and a 2D arcade-style fighting game [4]. However, the experience in such a field is still inadequate.

To allow innovative and efficient agents to emerge from this type of puzzle and provide insight into the design process and evaluation of these agents in an adaptive game environment, we choose one single game with dynamic rules as a research object and attempt to develop an intelligent agent for this game: *Baba Is Y'all*.

II. Background

Baba Is You is a puzzle game developed by Arvi Teikari originally for the 2017 Nordic Game Jam and then expanded to a full game with more mechanics. It has a similar game style like *Sokoban*, a player can control a character in the map to push blocks into a particular space to beat the level. However, some blocks may contain words and can form rules in the game. These rules can be created or broken at any time. Much of the game involves manipulating the rules in a certain order or orientation to allow the puzzle to be solved [1]. Fig.3 shows an example.

Besides the dynamic rule system, the most significant reason for choosing this game is that there has already been some flat research about this game, which offers some experience to study.

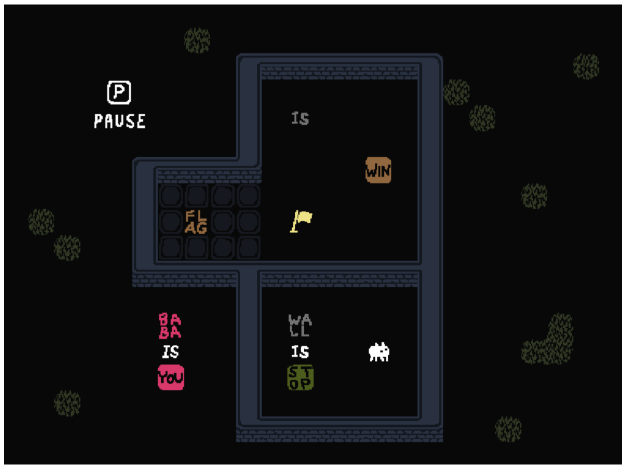


Fig. 3: A sample level in *Baba Is You*. Players have to break the active rule 'WALL-IS-STOP' and push the word blocks on the other side together, and create the rule 'FLAG-IS-WIN' to solve the level. [1]

A. *Baba is Y'all framework*

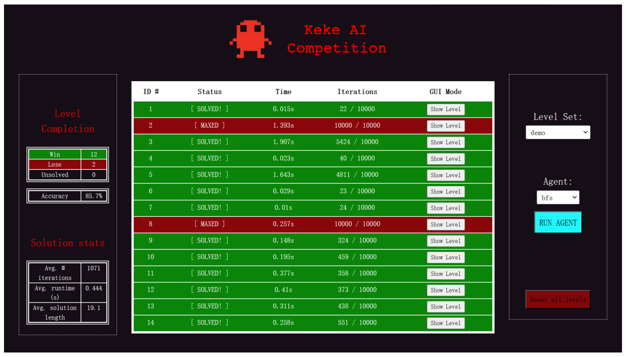
A level editor website called *Baba Is Y'all* allows players to make their levels, publish levels online, and rate levels. The high-performance levels in the website are good sources of test and debug material. We can also create our test maps to cover some extreme cases.

The integral version of *Baba Is You* contains hundreds of blocks and rules, which makes it hard to get started. This framework is the most original version instead of the latest version of the game. It only contains 11 object classes and 21 keyword classes and can form only nine formats of rules, which is a small subset of the full version of the game. Less complicated rules and core mechanisms make it the ideal material for algorithm design. [5]

B. *Keke AI Competition*

In IEEE Conference on Games 2022, a competition called "Keke AI Competition" adopts *Baba Is Y'all* framework and sets up an essential environment for programming.

The organizers reveal an open-source project containing a local website to evaluate the game [6]. The tests can fail or succeed, making it easy to verify the basic functionality of the agent and enter the improvement phase quickly. Some basic properties will also be shown on the website, like correctness rate, average time per solution, and the average length of solutions, all of which are significant figures for comparison and further improvement. One screenshot of the local evaluator is shown in Fig.4.



Win	12
Lose	2
Unsolved	0
Accuracy	85.7%

Avg. # iterations	1071
Avg. runtime (s)	0.444
Avg. solution length	19.1

ID #	Status	Time	Iterations	GUI Mode
1	[SOLVED!]	0.015s	22 / 10000	Show Level
2	[MAXED]	1.393s	10000 / 10000	Show Level
3	[SOLVED!]	1.907s	5424 / 10000	Show Level
4	[SOLVED!]	0.023s	40 / 10000	Show Level
5	[SOLVED!]	1.643s	4811 / 10000	Show Level
6	[SOLVED!]	0.029s	23 / 10000	Show Level
7	[SOLVED!]	0.01s	24 / 10000	Show Level
8	[MAXED]	0.257s	10000 / 10000	Show Level
9	[SOLVED!]	0.148s	324 / 10000	Show Level
10	[SOLVED!]	0.195s	459 / 10000	Show Level
11	[SOLVED!]	0.377s	358 / 10000	Show Level
12	[SOLVED!]	0.41s	373 / 10000	Show Level
13	[SOLVED!]	0.311s	438 / 10000	Show Level
14	[SOLVED!]	0.258s	551 / 10000	Show Level

Level Set: demo

Agent: bfs

Fig. 4: The evaluator screen of the offline Keke AI Competition offline interface

The project also provides a detailed GitHub wiki with a walk-through video to illustrate general ideas and variables in the environment, which saves time in reading and understanding code. Lastly, the organizers offer some valuable baseline agents in the project, which are the only agents for *Baba Is You* online and will be explained in IV.

III. Rules for *Baba Is Y'all*

Each level of the game initially contains particular objects and keywords, and players must push them to achieve winning conditions.

A. *Object and Keyword*

"Object" refers to the former word in the format "X-IS-Y"; the object can only be placed at the former for validation. Putting an object in the last place will not form a good rule.

"Keyword" refers to the latter word in the format "X-IS-Y", and the keyword can only be placed at the latter for validation. Similarly, putting the keyword in the former place will not form a good rule. Players can push all keywords under any rules.

Table.I shows all objects and keywords.

TABLE I: Objects and Keywords

Object	**Keyword**
BABA	YOU
BONE	WIN
FLAG	STOP
WALL	MOVE
GRAN	PUSH
LAVA	SINK
ROCK	KILL
FLOOR	HOT
KEKE	MELT
GOOP	IS
LOVE	

B. Rule formats

Rules for the level are defined by statements readable as English text and read from up, down, and left-right. So "X-IS-YOU" would be interpreted as a valid rule, but not "YOU-IS-X". As such, all rules must take one of these three formats [2] [7]:

- **X-IS-(KEYWORD)** where KEYWORD belongs to a word in the keyword class that manipulates the property of the game object class X (i.e., 'WIN', 'YOU', 'MOVE', etc.)
- **X-IS-Y** (X$\neq$Y) a transformative rule that changes all instances of sprites identified as X to the sprite Y.
- **X-IS-X** a reflexive rule that prevents any transformations occurring on the word block X. This differs from the previous rule X-IS-Y, as Y must differ from X for a transformation to occur. If a transformative rule is created, the X word block will not transform into Y with the reflexive tule active.

C. Detail Meaning

Table.II shows all detail keywords in the rules.

TABLE II: Objects and Keywords [1]

Rule Type	Definition
X	In this table refers to any object class
X-IS-X	objects of class X can't be changed to another class
X-IS-Y	objects of class X will transform to class Y, unless "X-IS-X" exists
X-IS-PUSH	X can be pushed
X-IS-MOVE	X will move towards its facing direction each action, X will turn around if meeting a wall. X will move whatever button the player press, even if SPACE is pressed.
X-IS-STOP	X will prevent the player from passing through it
X-IS-KILL	X will kill the player on contact
X-IS-SINK	X will erase any object contacting it, and will disappear with the contacted object
X-IS-HOT	X can only erase Y with "Y-IS-MELT", and will disappear with Y
X-IS-MELT	X can only erase Y with "Y-IS-HOT", and will disappear with Y
X-IS-YOU	All objects of X class are controlled by the player
X-IS-WIN	Player can win by touching X

D. Actions

Five actions are allowed to be performed in the game:

1) LEFT: all controlled objects move one step to the left, and all moveable objects move one step to their facing direction.
2) RIGHT: all controlled objects move one step to the right, and all moveable objects move one step to their facing direction.
3) UP: all controlled objects move one step up, and all moveable objects move one step to their facing direction.
4) DOWN: all controlled objects move one step down, and all moveable objects move one step to their facing direction.
5) SPACE: all moveable objects move one step to their facing direction, and **NO** controlled objects move.

E. Winning and Dead End

The winning condition is any player contacts any winning object, and the player still survives after contacting. Specifically, both conditions need to be satisfied at the same time:

1) 'X' in 'X-IS-YOU' contacts 'Y' in 'Y-IS-WIN'.
2) The number of 'X' must be greater than zero after contacting.

Since players must move to push words and interact with the environment, the game will enter a dead end if players cannot control anything on the map, and the player must restart the level in this case. To be more specific, the game will enter a deaf end if at least one of the following is satisfied:

1) No rule of format 'X-IS-YOU' exists.
2) The number of 'X' in 'X-IS-YOU' becomes zero.

IV. Baselines for *Baba Is Y'all*

Because of the timeliness and specificity of *Baba Is You*, there is almost no present algorithm discussing methods and ideas about its solution. The only well-performed algorithm is the best-first tree search solver in the game module of *Baba Is Y'all* framework, along with traditional search algorithms: BFS, DFS, and default agent - provided by organizers of "Keke AI Competition".

A. Default Agent

This algorithm is a best-first tree search algorithm and is first developed by *Baba Is Y'all* framework, which is used to evaluate the quality of a user-made level.

The algorithm uses a heuristic function based on the Manhattan distance to the centroid distance for three groups: keyword objects, objects associated with the 'WIN' rule, and objects associated with the 'PUSH' rule. These three categories are chosen because of their critical importance in solving process: winning objects are required to complete the level, keyword objects are responsible for rules manipulation, and pushable objects can directly or indirectly affect the layout of a level map and accessibility to reach winning objects.

The following equation represents the heuristic function:

$$h = \frac{n + w + p}{3}$$

The meaning of each variable is shown in Table.III.

The first baseline, the default agent in Keke AI Competition, originates from this algorithm. It is almost the same as the above best-first tree search. The only difference is that the default agent takes the average of all possible Manhattan distances in each category instead of the minimal nearest Manhattan distance.

The difference can be seen in Table.IV.

The pseudocode of the primary process is presented in Algorithm 1.

TABLE III: Variables in *Baba Is Y'all Framework*

Variable	Definition
h	The final heuristic value
n	The minimum Manhattan distance from any player object to the nearest winnable object
w	The minimum Manhattan distance from any player object to the nearest word block
p	The minimum Manhattan distance from any player object to the nearest pushable object

TABLE IV: Variables of default agent in *Keke AI Competition*

Variable	Definition
h	The final heuristic value
n	The average Manhattan distance from any player object to any winnable object
w	The average Manhattan distance from any player object to any word block
p	The average Manhattan distance from any player object to any pushable object

B. BFS / DFS Agent

As two of the most classic graph search algorithms, once the game board is represented as a graph, it can be explored using either BFS (breadth-first search) or DFS (depth-first search) and find a solution. BFS is typically better suited for finding the shortest path to a solution, while DFS is better for finding any solution with generally more calculation resources.

However, it is worth noting that the search space for "Baba is You" can be quite large, especially for more complex levels, so using heuristics or other optimizations may be necessary to find a solution in a reasonable amount of time.

Algorithm 1 Default Agent

```
Input: $init\_state$ - initial game state;
Output: $solution$ - action list $[a_0, ..., a_n]$ to win the game;
  function ITERSOLVE($init\_state$)
    $heap.push(newNode(init\_state))$
    while $heap$ is not empty do
      $curNode \leftarrow heap.pop()$
      $[score, children] \leftarrow getChildren(curNode)$
      for $child$ in $children$ do
        if $child$ win then
          return $child.actions$
        end if
        if $child$ died then
          $children.remove(child)$
        end if
      end for
      $heap.pushAll(children)$
      sort $heap$ in descending order by score
    end while
  end function
```

C. Random Agent

The algorithm will randomly choose one action from five possible legal actions, repeat the generation 50 times, and return an action list with a length of 50.

V. Agents based on Search Algorithm

Besides four baselines, we proposed two other agents based on two different search algorithms: Rolling Horizon Evolutionary Algorithms(RHEA) and Monte Carlo Tree Search(MCTS).

A. RHEA

RHEA is a subclass of Evolutionary Algorithm which evolve action sequences for games; it has the advantage of finding locally optimal solutions quickly based on the current state considering only a short period of future steps, which makes it ideal for single-player real-time games [8]. It has also been proved that RHEA performed well in general video game playing [9] [10].

The RHEA algorithm begins the game by initializing a population of P individuals, each representing a randomly generated action sequence of length L. These individuals are evaluated by simulating the action sequence, which means starting from the current game state and performing actions in sequence in order. The final game state reached is evaluated using a heuristic function, and the state value becomes the individual's fitness.

For each generation, the best E individuals(lowest fitness) are promoted directly to the next generation. The rest of the $P - E$ individuals are offspring generated by following procedures: for each offspring, K individuals will be chosen randomly from the previous generation, then the best two of them will become parents, and crossover is applied to create one offspring, which will mutate and be added into next generation. Then the next generation will replace the previous generation, and all individuals are evaluated as described above heuristic function, then this generation process repeats.

After reaching the appointed number of iterations or limitations of time, RHEA returns the first action in the best individual as the selected action to play in the game. For any subsequent game ticks, the previous final population won't be carried through to the next game tick.

The pseudocode of the RHEA process is described in Algorithm 2.

B. MCTS

Monte Carlo Tree Search(MCTS) is a heuristic algorithm with significant performance in many decision games. This algorithm does not require expert knowledge or specific problem rules of the game. Instead, it relies on simulation and statistical methods for decision-making. In general, the MCTS algorithm will start from the current game state, then choose a simulation direction with the highest discovery value and do plenty of simulations in this direction, then choose the action with the highest winning rate during simulation.

More specifically, the process of MCTS can be divided into four parts: selection, expansion, simulation, and backpropagation.

Algorithm 2 RHEA Agent

```
Input: $init\_state$ - initial game state;
Output: $solution$ - action list $[a_0, ..., a_n]$ to win the game;
  function ITERSOLVE($init\_state$)
      $cur\_node \leftarrow newNode(init\_state)$
      while $time$ and $iterations$ within limitations do
          $action \leftarrow ACT(cur\_node)$
          $actions.push(action)$
          $cur\_node \leftarrow nextNode(cur\_node, action)$
          if win then
              return $actions$
          end if
      end while
      return $actions$
  end function

  function ACT($cur\_state$)
      $pop \leftarrow init\_population(P)$
      $evaluate(pop, cur\_state)$
      while $time$ and $iterations$ within limitations do
          $next\_pop \leftarrow getBest(pop, E)$
          for i from 1 to $P - K$ do
              $candidates \leftarrow randomPick(pop, P - E)$
              $parents \leftarrow getBest(candidates, 2)$
              $new\_individual \leftarrow crossover(parents)$
              $new\_individual \leftarrow mutate(new\_individual)$
              $next\_pop.push(new\_individual)$
          end for
          $pop \leftarrow new\_pop$
      end while
      return $getBestAction(pop)$
  end function
```

Selection is the core idea of MCTS. The selection starts from the root node containing current game state information and checks its five children based on five legal actions. The purpose of this step is to find the child who is least discovered so that the algorithm can search uniformly like BFS; that is, the child that hasn't been expanded has higher priority than children that have already been expanded. So, on the one hand, if a child hasn't been explored, it can be chosen directly. Here we randomly choose an action among all children that haven't explored to import a random element. On the other hand, if all five children have been expanded, then the following evaluation function, Upper Confidence Bound Apply to Tree(UCT), will be implemented for all children, and the one with the maximal value will be chosen. The meaning of each variable is explained in Table V. Note that the UCT value will be greater if a child node is less explored than others.

$$UCT(i) = \frac{w_i}{n_i} + c\sqrt{\frac{\ln N_i}{n_i}}$$

The expansion is performed only when there's no child in the selected direction. In this case, the algorithm will perform the selected action based on the current node and store moved information in the child node.

TABLE V: Variables in UCT function

Variable	Definition
w_i	The number of winning times of child.
n_i	The number of simulation times of the child.
N_i	The number of simulation times of the current node.
c	A weight number.

The simulation is also called rollout or playout. Starting from the expanded(or selected) node, many random actions are chosen and performed on the node continuously until it reaches the winning or dead end. Then a heuristic function is implemented on the final state.

The simulation result will backpropagate along all of its parents. For each parent, the simulation times and the fitness value will be updated according to the heuristic value of the final state. Until then, an overall MCTS iteration is done.

The iteration of these four processes will repeat many times until reaching the ending condition or finding a correct answer. The pseudocode of while MCTS process is described in Algorithm 3.

Algorithm 3 MCTS Agent

```
Input: $init\_state$ - initial game state;
Output: $solution$ - action list $[a_0, ..., a_n]$ to win the game;
  function ITERSOLVE($init\_state$)
      $cur\_state \leftarrow init\_state$
      while $time$ and $iterations$ within limitations do
          $action \leftarrow ACT(cur\_state)$
          $actions.push(action)$
          $cur\_state \leftarrow nextState(cur\_state, action)$
          if win then
              return $actions$
          end if
      end while
      return $actions$
  end function

  function ACT($cur\_node$)
      while $time$ and $iterations$ within limitations do
          $action \leftarrow selection(cur\_node)$
          $new\_node \leftarrow expansion(cur\_node, action)$
          $fitness \leftarrow simulation(new\_node)$
          $backpropagation(newNode, fitness)$
      end while
      return $getBestAction(cur\_node)$
  end function
```

VI. Agents based on Reinforcement Learning

Besides search algorithms, reinforcement learning methods are also tried on this problem and relevant experiments are done. The policy algorithm implemented is Proximal Policy Optimization(PPO) [11].

A. Proximal Policy Optimization

PPO is a highly optimized policy algorithm based on the Policy Gradient algorithm. Based on Policy Gradient(PG), it added ”baseline”, ”discount factor”, ”advantage function”, ”importance sampling”, and other methods to improve efficiency and accuracy. It is one of the most widely used policy algorithms in reinforcement learning.

All terms used in this section are concluded in Table VI to clarify the description and explanation.

TABLE VI: Variables in PPO explanation

Variable	**Definition**
s	A state of the environment at a certain time.
a	An action taken to implement and will influence environment.
t	A certain point in the process of training and testing.
s_t	The state of environment at t point.
a_t	The action taken at t point after observing s_t.
π	The policy which can generate the possibilities of all available actions.
θ	The policy parameter to determine π.
π_θ	π determined by the policy parameter θ.
$p(s_t)$	The possibility that s_t occurs.
$p(s_{t+1}\|s_t, a_t)$	The possibility that s_{t+1} occurs in the condition that s_t occurs and a_t is taken.
$p_\theta(a_t\|s_t)$	The possibility that a_t is taken in the condition that s_t occurs in the current policy parameter θ.
τ	An sequence consists of s_t and a_t in every point t in a game period.
τ^n	A sequence τ identified by the number n.
$R(\tau)$	The total reward obtained in the sequence τ.
$R(\tau^n)$	The total reward obtained in the sequence τ identified by n.
r_t^n	The reward of a_t in τ^n.
$p_\theta(\tau)$	The possibility of the sequence τ occurs in current policy parameter θ.
$\bar{R}_\theta$	An unbiased estimate of the expected reward from one game under the current policy parameter θ.
∇f	The gradient of a certain function f.
$E_{\tau\sim p_\theta(\tau)}(x)$	The expectation of a certain variable x with respect to sequence τ in the distribution of $p_\theta(\tau)$.
γ	A discount factor that reduces the value of future rewards as the time horizon increases.
$A^\theta(s_t, a_t)$	Advantage function to measure how good it is if take a_t other than other actions at s_t, estimated by “critic”.
$V_\phi(s_t)$	The average of discounted rewards from s_t to the endgame.

1) Policy Gradient: In reinforcement learning, the agent will choose different actions to implement given the policy π; then, the environment will interact with the action(a), update itself, and return a new state as well as the reward to the agent. This process repeats until the game ends. The policy π is determined by a parameter called “policy parameter” denoted by θ. All the environment states and actions can be represented as a sequence τ. That is, suppose there are total T actions and states in the game period; the τ can be denoted as

$$\tau = \{s_1, a_1, s_2, a_2, ..., s_T, a_T\}$$

The probability of observing a specific sequence τ is defined as follows. Specifically, in the context of this problem implementation, given the level map as a fixed condition, the probability of the initial state $p(s_1)$ can be considered as 1 during the testing phase.

$$p_\theta(\tau) = p(s_1)\prod_{t=1}^{T} p_\theta(a_t|s_t)p(s_{t+1}|s_t, a_t)$$

The cumulative reward for an entire sequence τ throughout a game period is represented by $R(\tau)$. Given a policy defined by θ, the aggregate reward received in each game, governed by the distribution of θ, is characterized as the weighted sum of the reward across every sampled sequence τ. This can be computed as an unbiased approximation of the expected reward from a single game under the current policy parameter θ, which we designate as $\bar{R}_\theta$.

The objective of a single game is to maximize the reward $\bar{R}_\theta$. The natural idea is to take the derivative concerning $\bar{R}_\theta$ and use gradient descent to tune θ.

$$\bar{R}_\theta = \sum_\tau R(\tau)p_\theta(\tau) = E_{\tau\sim p_\theta(\tau)}[R(\tau)]$$

It’s important to note that $R(\tau)$ does not require differentiability and may even function as a black box. Thus, the gradient should be estimated using discretized sampled data. The following computation employs the property of the logarithmic function where $\nabla f(x) = f(x)\nabla \log f(x)$.

$$\begin{aligned}
\nabla \bar{R}_\theta &= \sum_\tau R(\tau)\nabla p_\theta(\tau) \\
&= \sum_\tau R(\tau)p_\theta(\tau)\frac{\nabla p_\theta(\tau)}{p_\theta(\tau)} \\
&= \sum_\tau R(\tau)p_\theta(\tau)\nabla \log p_\theta(\tau) \\
&= E_{\tau\sim p_\theta(\tau)}[R(\tau)\nabla \log p_\theta(\tau)] \\
&\approx \frac{1}{N}\sum_{n=1}^{N} R(\tau^n)\nabla \log p_\theta(\tau^n) \\
&= \frac{1}{N}\sum_{n=1}^{N}\sum_{t=1}^{T_n} R(\tau^n)\nabla \log p_\theta(a_t^n|s_t^n)
\end{aligned}$$

The above equations form the foundational process of Policy Gradient (PG): the agent interacts with the environment, updates the model via gradient descent using the experiences, and continuous the interaction in a recursive cycle until the end of the game. This workflow is visually represented in Figure5.

2) Baseline: The above PG implements the function of adjusting the model parameters according to the reward function. During the real training process, the reward function may not be as regular as we imagine, and cases where it diverges

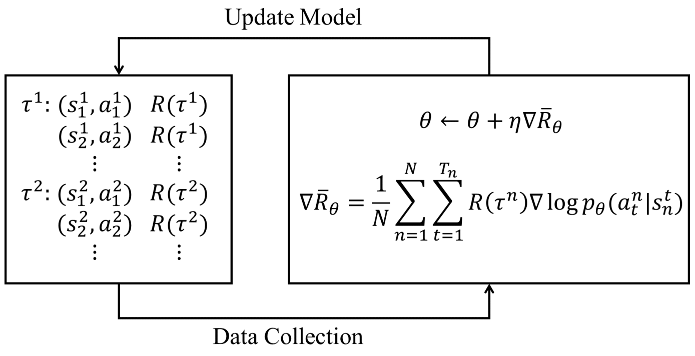

Update Model
τ^1: (s_1^1, a_1^1) $R(\tau^1)$
(s_2^1, a_2^1) $R(\tau^1)$
τ^2: (s_1^2, a_1^2) $R(\tau^2)$
(s_2^2, a_2^2) $R(\tau^2)$
$\theta \leftarrow \theta + \eta \nabla \bar{R}_\theta$
$\nabla \bar{R}_\theta = \frac{1}{N}\sum_{n=1}^{N}\sum_{t=1}^{T_n} R(\tau^n)\nabla \log p_\theta(a_t^n|s_n^t)$
Data Collection

Fig. 5: Flow of PG

significantly from zero can occur. Therefore regularization is necessary.

One way to achieve this is to add a baseline where the reward is positive and negative, and the baseline is the average of the rewards of all the sampled sequences. That is, replace the definition of previous $\nabla \bar{R}_\theta$ with the below one:

$$\nabla \bar{R}_\theta \approx \frac{1}{N}\sum_{n=1}^{N}\sum_{t=1}^{T_n}(R(\tau^n) - b)\nabla \log p_\theta(a_t^n|s_t^n)$$

where $b \approx E[R(\tau)]$.

3) Discount Factor: Not all rewards in the sequence have the same weight. Future rewards closer to the current point in time are apparently more valuable to the present. Therefore, we need to add a discount factor so that the future reward decays exponentially with the length of time since the present. That is, change the calculation of $R(\tau^n)$ into follows:

$$R(\tau^n) = \sum_{t'=t}^{T_n} \gamma^{t'-t} r_{t'}^n$$

4) Advantage Function: In the above PG, all pairs of state s_t and action a_t use the same reward function: the reward of this whole sequence $R(\tau)$. It should be replaced by a function related to s_t and a_t to improve accuracy. This function is called "advantage function", denoted by $A^\theta(s_t, a_t)$.

$$A^\theta(s_t, a_t) = \sum_{t'>t} \gamma^{t'-t} - V_\phi(s_t)$$

The first half $\sum_{t'>t} \gamma^{t'-t}$ represents the actual sampling discount reward. The second half $V_\phi(s_t)$, used to fit the discounted rewards from s_t to the endgame, is the average of the discounted rewards of all the sampled actions of s_t, provided by a neural network with the same structural policy but different parameter ϕ. The ϕ will also be updated together with θ in the training process to estimate better the average discounted rewards: the subsequent discounted reward for each action point in each data sequence is used as the feature to be learned, and the parameters are updated by minimizing the error between the prediction and the feature. The whole function represents the average advantage of taking a certain action a_t over other actions. A third-party "critic" gives this advantage.

5) Importance Sampling: Importance sampling is a statistical technique used to compute the expected value of a function $f(x)$, for a variable x following a probability distribution p. The most conventional method is to employ a random sampler conforming to the distribution p, to directly sample x multiple times, and subsequently calculate the average of its function value $f(x)$.

However, the distribution of p is typically unknown, thus the random sampler cannot be constructed and therefore making this approach inapplicable. There is an alternative feasible method for importance sampling: sampling from a known distribution q, determining the importance of the sample point by comparing and computing the probability of the sample point, and then calculating the mathematical expectation of the importance. Mathematically, both approaches can be proved to be identical.

$$\begin{aligned} E_{x\sim p}[f(x)] &= \int f(x)p(x)dx \\ &= \int f(x)\frac{p(x)}{q(x)}q(x)dx \\ &= E_{x\sim q}[f(x)\frac{p(x)}{q(x)}] \end{aligned}$$

After each parameter update of the above PG, it needs to re-interact with the environment to collect data, then update the model with the acquired data, and discard the data after the update; this data waste leads to slow network update speed. Therefore, it is necessary to reuse the collected data.

Suppose that the policy parameter used when collecting data is θ'. At this time, the collected data is stored in the memory bank. When there is enough data, the model is updated by the PG method above, and the policy parameter changes from θ' to θ after updating, which no longer matches the policy parameter of the collected data. Therefore, it is necessary to introduce importance sampling to correct the deviation between them. That is, replace the previous $\nabla \bar{R}_\theta$ as follows:

$$\nabla \bar{R}_\theta = E_{\tau\sim p_{\theta'}(\tau)}\left[\frac{p_\theta(\tau)}{p_{\theta'}(\tau)}R(\tau)\nabla \log p_\theta(\tau)\right]$$

However, this method is still primitive, so we need to introduce an advantage function to increase the fineness of the calculation, and the updated gradient becomes:

$$\nabla \bar{R} = E_{\tau\sim p_{\theta'}(\tau)}\left[\frac{p_\theta}{p_{\theta'}}A\right] = \sum_{t=1}^{T}\frac{p_\theta(a_t|s_t)}{p_{\theta'}(a_t|s_t)}A_t(s_t, a_t)$$

Although a has been used to estimate a through importance sampling, the gap between the two cannot be too large. Otherwise, it will lead to fallacy due to the limited number of samples. Therefore, a bias term called "KL divergence" must be introduced to penalize the distribution deviation between the two.

$$\nabla \bar{R} = \sum_{t=1}^{T}\frac{p_\theta(a_t|s_t)}{p_{\theta'}(a_t|s_t)}A_t(s_t, a_t) - \lambda KL[\theta, \theta']$$

This is the PPO's final version of $\bar{R}$.

6) Pseudocode: Considering all the above, there are three sets of parameters to update in PPO:

- Policy parameter θ: interacts with the environment to collect batch data, and then the batch data is associated with θ'. It will be updated every time.
- The copy of policy parameter θ': the association parameter of the data collected after the policy parameters interact with the environment, equivalent to the q distribution in importance sampling.
- Parameter ϕ of estimation V_ϕ: uses supervised learning to update its state evaluation based on the collected data and updates every time.

Then considering all other details, including loss function, backpropagation, and KL divergence calculation, the pseudocode of PPO is as follows:

Algorithm 4 Proximal Policy Optimization [12]

```
for $i \in \{1, \cdots, N\}$ do
    Run policy $\pi_\theta$ for $T$ timesteps, collecting $s_t, a_t, r_t$
    Estimate advantages $A^\theta = \sum_{t'>t} \gamma^{t'-t} r_{t'} - V_\phi(s_t)$
    $\theta_{old} \leftarrow \theta$
    for $j \in \{1, \cdots, M\}$ do
        $R_{PPO}(\theta) = \sum_{t=1}^{T} \frac{\pi_\theta(a_t|s_t)}{\pi_{old}(a_t|s_t)} A^\theta - \lambda \mathrm{KL}[\pi_{old}|\pi_\theta]$
        Update $\theta$ by a gradient method w.r.t. $R_{PPO}(\theta)$
    end for
    for $j \in \{1, \cdots, B\}$ do
        $L_{BL}(\phi) = -\sum_{t=1}^{T} (\sum_{t'>t} \gamma^{t'-t} r_{t'} - V_\phi(s_t))^2$
        Update $\phi$ by a gradient method w.r.t. $L_{BL}(\phi)$
    end for
    if $\mathrm{KL}[\pi_{old}|\pi_\theta] > \beta_{high} \mathrm{KL}_{target}$ then
        $\lambda \leftarrow \alpha\lambda$
    else if $\mathrm{KL}[\pi_{old}|\pi_\theta] < \beta_{low} \mathrm{KL}_{target}$ then
        $\lambda \leftarrow \lambda/\alpha$
    end if
end for
```

B. Reward Function

In addition to the updated policy PPO, another equally important factor is the evaluation function applied to this problem, which calculates r_t^n in PPO. Here two reward functions are provided in this problem.

As mentioned before, the organizer of Keke AI Competition provides a baseline heuristic function $h = \frac{n+w+p}{3}$ where n, w, p are the average Manhattan distance from any player object to any winnable object, word block, pushable object respectively. This function can also be used as the reward function in the environment of PPO.

The second reward function is based on domain knowledge. Although not a full Version of the game, James Version still has the most basic game mechanics. Therefore, the strategies and tactics specified for the full game still apply to the James Version. Based on the discussion and summary of domestic and foreign game forums, the second reward function adopts the idea of classification, uses different calculation strategies for different impact factors to calculate the reward value respectively, and gets the total reward value after weighted summation.

From a calculation strategy's perspective, how each factor's reward value is calculated will be introduced below.

1) Existing Numbers: Consider a class C that contains several single individual objects. Suppose the class in the point t is C_t, and in the previous point $t-1$ is C_{t-1}, and the number of individuals is represented as $len(C)$. One way to calculate reward is by assigning a weight value w_c for every available individual in class C and using the difference between two adjacent time points as the reward for the latter point. That is,

$$r_{num}(t; C) = w_C \times [len(C_t) - len(C_{t-1})]$$

This is the simplest reward function, both intuitively and analytically. The class using this is the most basic game and does not have many complicated rules.

The type of objects using this calculation formula is "player" and "pushable object".

2) Compose and Destroy: Rules are one of the most significant parts of the game; it is a general agreement that composing rules is the key to solving a level. So it should be encouraged to compose the rules with positive rewards and not encouraged to destroy the rules with negative rewards.

A level usually has several already composed rules and a few scattered word blocks. Most of the time, players need to use almost all the scattered word blocks to form new rules or move under the existing rule system, and the behavior of breaking the existing rules is relatively rare. Even if there is, it only breaks one or two key rules, and most of the existing rules remain unchanged. Thus, although again encouraging, the encouragement to compose the new rule should be slightly larger than the magnitude of the encouragement to destroy the existing rule.

Suppose the reward for compose of new rules is w_{comp} and for destroying of old rules is w_{des}, the number of compose and destroy rules are n_{comp} and n_{des} then the reward for rules is:

$$r_{rule}(t) = w_{comp} * num_{comp} - w_{des} * num_{des}$$

where $w_{comp} > w_{des}$.

This calculation formula is only used for "rules".

3) Weighted Distance: The reward function is suitable for evaluating most of the two types of distance-related objects.

For two classes of objects, the distance between each distinct pair of objects first needs to be calculated and then sorted. For a distance at a certain rank, assign a weight to each unit distance of the distance. Since the distance at the top of the list is more important than the latter, we need a decreasing factor to decrease the weight per unit distance as the rank increases. The unit distance weight of each rank multiplied by the distance of that rank is the reward value obtained for that rank. The sum of the reward values of all positions is the total reward value of the two types of objects.

Although it may not seem obvious, the decreasing factor is effective, as evidenced by subsequent experimental results.

Specifically, suppose the weight of rank 1 is w_0, the decrease factor is α, the distance of rank i is d_i, then the total reward r_{dist} for these two classes is

$$r_{dist} = \sum_{i}^{i \in rank} w_0 q^{i-1} d_i$$

A mathematical analysis of the above equation shows that $w_0 < 0$ should be satisfied if we want to encourage the two types of objects to move closer to each other. If we want to encourage the two types of objects to move away from each other, we need to satisfy $w_0 > 0$.

The detailed pseudocode of the calculation is shown in Algorithm. 5.

Algorithm 5 Calculate reward by weighted distance

```
Input: $C_1$ - first class, $C_2$ - second class,
       $w_0$ - initial weight, $\alpha$ - decrease speed;
Output: $reward$ - reward by weighted distance;
  function REWARDBYWEIGHTEDDISTANCE($C_1, C_2, w_0, \alpha$)
      Initialize $distances$ as empty list
      for each $i$ in $C_1$ do
          for each $j$ in $C_2$ do
              Append distance($i$, $j$) to $distances$
          end for
      end for
      Sort $distances$ in ascending order
      $reward \leftarrow 0$
      $weight \leftarrow w_0$
      for each $distance$ in $distances$ do
          $reward \leftarrow reward + weight \times (distance + 0.1)$
          $weight \leftarrow weight \times \alpha$
      end for
      return $reward$
  end function
```

Note that $weight \times (distance + 0.1)$ is used instead of $wight \times distance$ to avoid gradient explosion.

The pair of objects using this calculation formula are listed in Table VII.

TABLE VII: Pairs calculated by Algorithm 5

Class 1	**Class 2**
player	killer
player	sinker
pushable	sinker
player	winnable
player	word

4) Tuning of parameters: A unique program was written to adjust the parameters because it is difficult to directly determine the weights' values and decreasing factors' values. The input of this program is a level map and action instructions, and the output is all the reward value details corresponding to the action instructions. By manually selecting levels and entering instructions, it is possible to observe whether the reward value of a specific behavior is as expected and thus fine-tune the parameters of different parts. As the difficulty of the selected level rises, the parameters will be closer and closer to the actual value.

VII. Experiments

This experiment runs all algorithms on all levels, as the same rule in Keke AI Competition. The ranking priority is to look at the winning rate first and then $1/(Avg.runtime \times Avg.length)$.

A. Algorithms

Eight algorithms are tested in the experiment, including four baselines(default, BFS, DFS, random), two search algorithms (RHEA, MCTS), one reinforcement learning algorithm, and the winner algorithm of Keke AI Competition in 2022 [13].

All algorithms except reinforcement learning using heuristic function adopt the same function as described in Section IV-A, that is, $h = \frac{n+w+p}{3}$, where n is the average Manhattan distance from any player object to any winnable object, w is average to any word block, and p is average to any pushable object.

Mainly, for the MCTS algorithm, a multiple-beginning strategy is used. In the result below, 'MCTS-10' refers to the fact that this MCTS version has ten beginnings, and each is separated from others, 'MCTS-1' refers to the original version, which has only one beginning to search.

For reinforcement learning, three versions are compared in the experiment. All three versions use PPO as a strategy but with different reward functions. "RL" is the most original algorithm; it takes the heuristic function $h = \frac{n+w+p}{3}$ as the reward function directly. "RL-d" uses the reward function described in section VI-B. The values of the parameter after tuning are listed in Table VIII, IX, and X. "RL-0" use the same weight value as "RL-d" with all decrease factor to be 0; this is to compare with "RL-d" to confirm the validation of the decrease factor.

TABLE VIII: Variables in Existing-Number strategy

Class C	**Individual Weight w_C**
players	1
pushables	1

TABLE IX: Variables in Compose-Destroy strategy

Class C	**Compose Weight w_{comp}**	**Destroy Weight w_{des}**
rules	3	2

B. Dataset

Although Keke AI Competition offers some demo levels for testing [1], they are too small as a dataset for experiments.

As referred before, *Baba Is Y'all* is an online level editor website allowing players to upload their own created levels. This website also offers download functionality to obtain all levels in JSON format. This dataset is approximately 250 levels, enough for the experiment.

TABLE X: Variables in Weighted-Distance strategy

Class 1 C_1	**Class 2 C_2**	**Initial Weight w_0**	**Decrease Factor α**
players	killers	2.5	0.85
players	sinkers	2.5	0.85
pushables	sinkers	-3	0.85
players	winnables	-5	0.9
players	words	-1	0.8

C. Result

Referring to Keke AI Competition, all algorithms have the constraint of 10,000 iterations and 10.0s. The experiment results are shown in Table.XI. The meanings of indices are as follows:

1) Agents: the algorithm for the agent.
2) Winning Rate: the percentage of successfully solved levels to total levels.
3) Avg. runtime (s): the average time spent per level, including unresolved levels, in seconds.
4) Avg. solution length: the average length of the solution for each level, including unsolved levels.
5) Rank: rank under the rules of Keke AI Competition in 2022

TABLE XI: Experimental Result

Agents	**Winning Rate**	**Avg. runtime (s)**	**Avg. solution length**	**Rank**
MCTS-1	79.30%	2.464	99.8	1
MCTS-10	78.50%	2.450	94.8	2
2022 Best	65.40%	4.028	93.2	3
Random	59.80%	1.176	50.0	4
Default	57.70%	4.484	15.3	5
BFS	56.10%	4.518	25.9	6
DFS	48.2%	5.506	332.4	7
RHEA	29.3%	4.689	8.6	8
RL	10.57%	6.27	62.93	9
RL-d	9.76%	6.09	77.8	10
RL-0	8.13%	6.54	83.48	11

VIII. Analysis

A. Baselines

All four baselines show correct properties. Random does not include the search process, so it has a minimal runtime for each level, and its solution length is all 50.0. DFS regards depth as the highest priority, so it has the maximal solution length among all algorithms. In comparison, BFS regards breadth as the highest priority, so its solution length is much less than DFS. The default agent is the only algorithm with a heuristic function among the four baselines, so it has the highest winning rate.

B. MCTS

MCTS reaches the highest winning rate among all algorithms. The multiple advantage properties of MCTS can explain this.

Compared to traditional search algorithms, MCTS is more efficient. It evaluates the value of each action by simulating random games rather than searching the entire game tree. This scalability and computational efficiency make MCTS particularly effective in handling complex problems like “Baba Is You”.

Another reason is that MCTS has a stronger exploration ability. It employs random simulation to explore different decision paths extensively. This allows it to perform well in complex, unknown, or uncertain environments and discover better solutions than the four baselines.

C. Multiple-beginning MCTS

In the above result, the MCTS algorithm using and not using multiple beginnings have almost the same winning rate. After many other experiments, it can be concluded that there is no obvious difference between these two types, and the tiny difference in winning rate can be regarded as an error within the acceptable range.

One possible explanation is that even for MCTS with multiple starting points, each still uses the same algorithm. The advantage of multiple starting points over single starting points is that they can be simulated more times at the same depth and are more likely to achieve the optimal value at the same depth. For the rules of the game Baba Is You, the single-beginning MCTS is already very close to the optimal value, so multiple-beginning is no longer an advantage.

D. RHEA and MCTS

It is observed that the results of RHEA and MCTS differ greatly for the same search type of algorithm. One possible explanation is that RHEA, which applies to general video games, does not apply to puzzle games.

RHEA considers only short-time sequences of future actions, which makes it less considerate, more efficient to solve, and very real-time.RHEA also bases its decisions on the current environment’s changing state, allowing it to explore solutions that apply to the current state. Both of these factors make it suitable for use in real-time fighting games.

However, puzzle games emphasize solving results rather than immediacy, so RHEA’s greatest advantage no longer exists.

In the iterative process, RHEA introduces new individuals randomly and then detects whether the unique individuals apply to the current state. There is almost no connection between the two individuals. Despite crossover and mutation, the parents of the crossover are still randomly generated, so the relationship between the two individuals is minimal.

MCTS, on the other hand, first determines all possible actions in the current state and then allocates the same search resources to each step based on these actions as much as possible. This search type is more systematic, comprehensive, and uniform. Combined with the 10-second and 10,000-iteration limits, RHEA does not explore enough random solutions in a limited time, so MCTS is far more accurate than RHEA.

E. Reinforcement Learning

The best performance of the reward function is the one provided by the competition organizer may result from the fact that the organizer selected a function with both applicability and quality after testing to provide a higher-quality function to the participants. The reward function containing a descent factor, on the other hand, relied mainly on domain knowledge. Although it was feasible, its high quality could not be guaranteed and thus had a lower winning rate.

It can be observed that the average runtime of all three algorithms of reinforcement learning is higher than the others. This is because the environment used for reinforcement learning is Python, but the problem environment for Keke Is You is JavaScript, which requires calling JavaScript wrapper functions from Python each time the action is executed. This cross-language call is much less efficient than calling the function within the same language, thus elongating the runtime.

F. Reinforcement Learning and MCTS

In the experimental results, all three reinforcement learning algorithms performed poorly in winning rate, significantly surpassing MCTS. This could be attributed to the inherent nature of deep learning itself and the specific characteristics of the Keke Is You problem: trade-off between exploration and exploitation, sparse Rewards, long-term Planning, and computational Resources.

In any reinforcement learning environment, the trade-off between exploration (trying new actions to gather more information) and exploitation (selecting the best action based on known information) is an important consideration. PPO relies on its policy update rules to balance exploration and exploitation. On the other hand, MCTS achieves a natural trade-off between exploration and exploitation through its UCT policy, which has demonstrated powerful performance in many board games like Go. In this game, where the solution space is vast, effective exploration can be crucial, potentially giving MCTS an advantage.

Regarding rewards, rewards in the “Keke is You” game are often sparse, only obtained upon solving puzzles. Handling sparse rewards is a challenge for policy gradient-based methods like PPO. MCTS, on the other hand, does not rely on reward signals for search and may handle this situation better.

Also, “Keke is You” typically requires a series of actions to solve puzzles, necessitating long-term planning by the agent. While PPO can perform some planning by training a useful value function, MCTS explicitly performs long-term planning by constructing a search tree, which may be more effective in this case.

PPO requires many samples for computational resources to learn effective policies, which can demand significant computational resources and time. On the other hand, MCTS is a search-based method that can perform effective searches with limited computational resources. Although the search depth limits the efficiency and MCTS may face a vast search space and high computational costs for very complex problems, “Keke Is You” problem is still in the region that MCTS could handle.

G. Further Exploration

Since the reinforcement learning effect did not meet expectations, we further tested the performance effect at different difficulty levels.

All levels were divided by the number of steps of answers, and the relationship between difficulty and the number of steps is in Table XII.

TABLE XII: Level difficulty

Algorithm	Winning Rate
1	< 5
2	$6 \sim 10$
3	$11 \sim 20$
4	$21 \sim 40$
5	> 40

The results of the test with a difficulty of 1 are shown in Table XIII.

TABLE XIII: Result in Difficulty 1

Agents	Winning Rate	Avg. runtime (s)	Avg. solution length
RL	43.53%	3.72	38.21
RL-0	41.18%	3.38	43.88
RL-d	47.65%	3.44	44.93

The experimental results indicate that RL-d with a decreasing factor achieved the highest winning rate, followed by RL with the default reward function, and RL-0 without a decreasing factor had the lowest winning rate.

The specificity of the training set can explain this phenomenon. Difficulty level 1 represents solutions that require five or fewer steps. Within this range, two situations exist: the first is that the level is indeed very simple, with only a few word blocks, requiring only a few steps to achieve victory. The second is when the level contains a large number of word blocks, but almost all of them are distractors, and winning only requires moving in one or two directions. This situation tests the reward function’s resistance to interference from word blocks, a strength of RL-d with a decreasing factor.

Due to the constraint of “solutions with five or fewer steps”, all levels favoring RL-d are retained, while most of the longer-step levels are filtered out. This results in a higher proportion of these levels in difficulty level 1 than all. Therefore, RL-d has an advantage under the constraint of difficulty level 1, surpassing the accuracy of RL using the original reward function. RL-0, on the other hand, has the lowest winning rate because it does not utilize a decreasing factor.

Only experiments with difficulty level 1 are conducted here, and due to time and computational limitations, experiments for other difficulty levels could not be completed within this semester.

IX. Conclusion and Future Improvement

The demand for intelligent agents arises as the puzzle's complexity keeps increasing. Although many algorithms perform well in traditional puzzles with static rules, dynamic rule puzzles remain to be explored. Future tests on one of the latest games, , will conduct based on the programming environment provided by Keke AI Competition.

In the first stage, we clarified the game's rules, finished configuring the game simulation environment, and successfully ran several of the given baselines to prove it.

In the second stage, we implemented two search category algorithms, RHEA and MCTS, and tested them in an expanded test set. Among them, RHEA performed poorly because it did not apply to this game, while MCTS outperformed any other algorithm, including the competition's winner of last year.

In the third stage, we implemented reinforcement learning with three different reward functions and tested them on the same set as the second stage. All three are at the bottom of the ranking, while the default reward function shows an advantage over the other two.

In future work, it could be worthwhile to investigate the use of Proximal Policy Optimization (PPO) to learn a heuristic policy and value function that could guide the search in Monte Carlo Tree Search (MCTS). The combination of these two methods holds the potential of achieving the strengths of both: the sample efficiency and the capacity to generalize from PPO, and the powerful search capabilities of MCTS.

References

[1] J. T. M Charity, Sarah Chen, "Keke AI competition," http://keke-ai-competition.com/, 2022.

[2] G. N. Yannakakis and J. Togelius, *Artificial intelligence and games*. Springer, 2018, vol. 2.

[3] R. Canaan, H. Shen, R. Torrado, J. Togelius, A. Nealen, and S. Menzel, "Evolving agents for the hanabi 2018 cig competition," in *2018 IEEE Conference on Computational Intelligence and Games (CIG)*. IEEE, 2018, pp. 1–8.

[4] F. Lu, K. Yamamoto, L. H. Nomura, S. Mizuno, Y. Lee, and R. Thawonmas, "Fighting game artificial intelligence competition platform," in *2013 IEEE 2nd Global Conference on Consumer Electronics (GCCE)*. IEEE, 2013, pp. 320–323.

[5] M. Charity, A. Khalifa, and J. Togelius, "Baba is y'all," http://equius.gil.engineering.nyu.edu/, 2022.

[6] S. Bozyel, "Keke AI competition," https://github.com/sonelb/Keke-AI/blob/main/agents/random_BABA_GO_solvesRiverone_AGENT.js, 2022.

[7] M. Charity, A. Khalifa, and J. Togelius, "Baba is y'all: Collaborative mixed-initiative level design," in *2020 IEEE Conference on Games (CoG)*. IEEE, 2020, pp. 542–549.

[8] D. Perez, S. Samothrakis, S. Lucas, and P. Rohlfshagen, "Rolling horizon evolution versus tree search for navigation in single-player real-time games," in *Proceedings of the 15th Annual Conference on Genetic and Evolutionary Computation*, ser. GECCO '13. New York, NY, USA: Association for Computing Machinery, 2013, p. 351–358. [Online]. Available: https://doi.org/10.1145/2463372.2463413

[9] R. D. Gaina, D. Perez-Liebana, S. M. Lucas, C. F. Sironi, and M. H. Winands, "Self-adaptive rolling horizon evolutionary algorithms for general video game playing," in *2020 IEEE Conference on Games (CoG)*, 2020, pp. 367–374.

[10] R. D. Gaina, S. Devlin, S. M. Lucas, and D. Perez-Liebana, "Rolling horizon evolutionary algorithms for general video game playing," *IEEE Transactions on Games*, vol. 14, no. 2, pp. 232–242, 2022.

[11] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, "Proximal policy optimization algorithms," 07 2017.

[12] N. Heess, D. TB, S. Sriram, J. Lemmon, J. Merel, G. Wayne, Y. Tassa, T. Erez, Z. Wang, S. Eslami *et al.*, "Emergence of locomotion behaviours in rich environments," *arXiv preprint arXiv:1707.02286*, 2017.

[13] J. T. M Charity, Sarah Chen, "Keke AI competition," https://github.com/MasterMilkX/KekeCompetition, 2022.