CS6534 Guided Study

Final Report

Multidimensional Decision Making using MCTS

City University of Hong Kong

Department of Computer Science

ZHAI Guanxun

54345382

Index

Introduction

As the Artificial Intelligence Technology develops, more generalized AI implementation methods are required to be able to solve problems under different application environments. In computer game discipline (including AI for actual board games, like Go), the most frequent used algorithm nowadays is the Monte-Carlo Tree Search (MCTS), a domain-knowledge free method with stochastic simulation. This method has been implemented on some Go agents and achieve remarkable success. In last year, the game between Google's Go agent AlphaGo and Korean Go player Sedol Lee has proved that with sufficient computing resources and deep learning implementation, MCTS and its modified algorithm can beat human players in almost all kinds of games.

Nowadays, most of AI programming in computer games are behavior scripting. Strictly speaking, such implementations cannot be considered as "intelligent". Some researchers suggest that if we could build AI agents that can operate on different games, then it is practical to achieve higher-level AI. Thus, General Game Playing AI are proposed, referring to those agents that can compete or beat human players in different games without implementing domain-knowledge of specific games. To encourage better AI designing, since 2005, AAAI held an annual competition for general game playing agents [6]. The target of general game playing is not limited to board games. Video games and games with incomplete information are also included. University of Essex also holds a competition named GVG-AI competition for generalized video game playing [11]. As MCTS is a basically a stochastic method, it can gradually generate solution to different kinds of computer games. Due to this reason, it is very frequently used in general-purpose

AI agents. Practically there are too many genres of games to be abstracted using the same model. Thus, the Game Description Language (GDL) [13] was developed for transforming game rules into programmable form. The original version of GDL only consider complete information games, which is referred to as GDL-I. For GDL-II, the

incomplete information games are also considered [4]. Literally, GDL is a set of definitions for terms that can beat generalize games. Terms within the definition can also be considered.

As video game industry develops, more complex games with different sets of game rules become popular. To construct AI for these games, it is necessary that limit the AI function into specific sub-system. In real-time strategy games like StarCraft or table-top games like Settlers of Catan [14], how to build the best set of units using limited resources is players' focus. For such computer games or table-top games, resource management are critical to win the game. It is thus meaningful if a sub-system AI for resource management can be implemented in those games. To build a generalized AI agents for these games, we may consider construct a partial AI for part of the game rules. Normally, such games have full information. However, as number of players or available units increase, the decision would be more complex considering that each unit might have complex functions. And how players control the units is even more undetermined. As a stochastic method, MCTS is a proper way to achieve such objectives. Even that in these games, controlling the units is more important. But such gameplay requires real-time AI agents, it might not be easy to build a search tree for such a problem because the position might change frequently. Here we will only try to propose a method for how to choose proper units to build, which is a more discrete problem. This can be considered as a partial approach to a general AI agents for a specific genre of computer games.

To further illustrate the objective, we may refer to some AI designing using MCTS. In real-time Strategy games, MCTS is also applicable. However, the problem of designing AI using MCTS is that in some implementations, the AI tends to exhaust the resource quickly and is enable to optimize the game decisions. Normally in the computer game mechanisms, a balance between exploiting resource and exploiting functional units is critical. In those cases, where AI exhaust resource fast, the functional units' overall value will be no match to the human player as game process. One possible reason is the value evaluation is biased. To improve the process, the associated decision based on different values might be better to conduct in the AI

implementation. The most frequently used techniques to improve is the Upper Confidence Bound Tree-Search (i.e. UCT). And in this study work, this implementation is applied to study how to improve MCTS AI in strategy game resource management.

I. Objective

The objective of this study is to apply modified MCTS algorithm on potential computer games, especially for the games which require resource management. The basic idea is to expand the frequently used UCT algorithm's value function in the selection step of MCTS. By constructing multiple search tree, each corresponds to one discrete-value attribute to represents the resource of the player, the learning process might be possible for different games. And the performance will be evaluated based on the how the total value of the units is maximized with limited resources in a finite time domain.

As a guided study topic, this work will use abstract game models to simulate how real games work. The simulation will be simple as some basic units with limited attributes. The idea is to construct a basic model of game AI. Hopefully, the game model can represent how resource modules work in different games. The abstraction will be generated mainly from real-time strategy games like Warcraft. Also, modern tabletop games, like Settlers of Catan will also be considered as a good reference. Normally these games vary greatly from each other regarding its theme. The idea is to generalize the game mechanism and use MCTS algorithm to test how this algorithm can be applied on the mechanism

II. Related Work

1. Monte-Carlo Tree Search & Upper Confidence Bounds of Trees [1]

Monte-Carlo Tree Search (MCTS) is one of the mostly used and effective algorithm for game playing. MCTS algorithm normally is used to build up a decision search tree by random samples and updated gain value through back-propagation. This method

can be used both in deterministic and stochastic games. Due to its good scalability, many researchers choose MCTS to design AI agents for GO and has successive results. The chess-like games, which have more biased rules, also can have a good effect with some domain knowledge as backup for the search policy. MCTS contains 4 steps: selection, expansion, simulation and back-propagation.

Selection: Starting at the root node, searching down the tree to the child node using tree policy. Stop at the node with the best reward.

Expansion: If the node is not terminal and has never been visited previously, expand this node by adding children to it.

Simulation: Simulate the game from this node by the default policy. In the simplest case, the default policy is just randomly choosing actions from the expand node.

Back-propagation: After simulating to a terminal state, use the terminal result as the back-propagation value and update all the previous nodes visited based on this result.

Another term used frequently about MCTS is “playout”. It is another expression of simulation, meaning that using default policy to complete the task. The process after expansion are completely irrelevant to the tree policy. Like Monte-Carlo Method, this process is often stochastic. Many variation of MCTS focus on the tree policy modification. And some modify the default policy of simulation.

Also, there are some characteristics about MCTS that explains why it’s a popular choice: The first is its domain-irrelevance. MCTS doesn’t require specified domain knowledge for heuristic programming. As due to its stochastic default policy in simulation phase, the iteration can be faster comparing with those methods with heuristic programming using domain specific knowledge. An extreme case is when we want to design agents for a brand-new game, if we cannot have enough resources (time, labor, budgets) to test a game and to gather knowledge to guide the learning process, a stochastic approach can be a simple solution to have some acceptable result. second is continuing updates due to the back-propagation. Such steps guarantee that the nodes’ data is always up-to-date, which means flexible reactions to the current

game status. The 3rd is asymmetric convergences. As the game status keep updating, the terminal states will emerge and for some games, the tree might gradually converge to some frequently visited nodes. Such process may help analysts to generate some knowledge of the game. For instance, we may use shape analysis for the tree and recognize some features using pattern recognition techniques. The results can be extracted and used for further analysis. Such characteristic can be used in game designing and game prototyping.

Among them, Upper Confidence Bounds of Trees(UCT) is the most effective method, combining MCTS with Upper Confidence Bounds(UCB). It considers how to balance exploration and exploitation to gain a total better result. The typical problem of exploration and exploitation is the bandit machine problem. Here we simply describe this problem: Suppose we have a bandit machine with multiple arms. Some of them have higher chance to get high score. And we want to get the highest score in a finite number of attempts. Because the process is totally random, we can only get the chance distribution by swing the arms. And these attempts will contribute to the total score. The UCB algorithm uses the sum of two terms for determining which arm to choose. The simplest form(UCB1) is as follows:

$$UCB1 = \overline{X}_j + \sqrt{\frac{2\ln n}{n_j}}$$

And this form is adapted by UCT algorithm, adapting form of UCB.

$$UCT = \overline{X}_j + 2C_p\sqrt{\frac{2\ln n}{n_j}}$$

Here *n* denotes the number of times the current (parent) node has been visited, n_j is the number of times child j has been visited and C_p is the constant for tuning. The left term is for exploitation, which is the upper boundary. While the term on the right is for exploration. When the node is rarely visited, the denominator will be small, and there is high chance that it can be visited if the total visit is frequent. These situations suggest that after many attempts, this unvisited arm is very worth a try because it is possible to be a higher reward arm. However, if the average gain turns out to be small, then it will gradually be ignored after enough attempts. This is a very simple expression,

yet it's good enough to show how to solve the bandit problem, which means a balance between exploitation and exploration. This is now frequently used in MCTS as tree policy. And this study is trying to propose how to modify the tree policy based this simple expression of UCT.

2. General Game Playing and Game Description Language [13]

Another research target is the General Game Playing(GGP), presented by a Stanford game research team. This idea is closely related to the Metagame theory which is proposed by Berney Pell [12]. In a nutshell, GGP means the ability to play different games using the same agent. To achieve this target, we need to have a general syntax to describe different games. Here the researchers use Game Description Language.

The Game Description Language is about how to generalize games in an abstract way. Normally a game will include these phases: Starting the game, Action of one player, Legal Verification, turn shifting and Terminal State Checking. Markov decision process is widely applied in game research, which define decisions to be a sequence of states and actions. Adapting this feather, the research team denote GDL as following keywords [2] [13]:

`Distinct`: This predicate is used to require that two terms be syntactically different.

`Does`: The predicate `does(?r,?m)` means that player (or *role*) `?r` makes move `?m` in the current game state.

`Goal`: The predicate `goal(?r,?n)` is used to define goal value `?n` (usually a natural number between 0 and 100) for role `?r` in the current state.

`Init`: This predicate refers to a fact about the initial game state.

`Legal`: The predicate `legal(?r,?m)` means that `?m` is a legal move for role `?r` in the current state.

`Next`: This predicate refers to a true fact about the next game state.

`Role`: This predicate is used to add the name of a player.

`Terminal`: This predicate means that the current state is terminal.

`True:` This predicate refers to a fact about the current game state.

These keywords are defined with respect to the definition given by this research team. The content can be expanded or limited under different circumstances. We'll talk about this later the topic of non-deterministic games adaptation to the GDL, GDL-II. Now GGP is referred to as an annual competition held by AAAI's summer conference. This competition is aimed at promoting the research in this area. There are typical competing subjects (i.e. games) are Tic-Tac-Toe, chess, maze search etc.

There is a problem on the GGP's scale. It seems impossible to build an ideal GGP agent for all kinds of games, considering the games vary greatly in different genres. Some are deterministic games as some have random results. Such facts will have diverse or even opposite policies and strategies. So, aiming at a GGP method for a specific domain is more practical.

3. Generalized Monte-Carlo Tree Search & UCT enhancement for GGP

As for GGP agents, how to optimize the reward without domain-specified knowledge is one focus. For generalizing purpose, optimizing on algorithm alone is an approach to how to strengthen the performance. Considering that basic UCT form is simple, when applying it on different situations some flaw will emerge. From the research of Hilmar Finnsson, UCT is locally operated in many situations. This will consequently ignore some already explored good nodes on the fringe of the whole tree [4]. Such behavior might be a waste considering the situation that a good node has already been found but ignored until most unexplored nodes are visited. Instead, if the agent can take the good already-explored nodes as priority and exploit early, the performance will be better in a limited number of iterations.

Another potential optimization method is about how to eliminate ineffective patterns when simulating the game. These features are normally some repeating process, like in a chess game, both players are exhausted in pieces, they might take advantage of the rule and reach a dead-lock situation where both players cannot win the game. In some cases, such a situation can be regarded as a draw. However, in the view of trying to better the opponents, this should be avoided. Another kind of

features that is needed to be removed are unstable features. There are some GGP agents using learning techniques and generate features which can be used to speed up the simulation. These features might just be fake if they happened to emerge in limited iterations while in actual playouts they barely emerge.

There are two methods trying to solve this problem: Unexplored Action Urgency and Early Cutoffs [4], presented by this research paper. Unexplored Action Urgency is overall a modification of UCT algorithm. While Early Cutoffs operate on general idea of MCTS.

Unexplored Action Urgency distinguish unexplored and explored nodes first. By comparing their best performance of explored nodes and urgency of unexplored nodes, the algorithm will have a biased strategy encouraging more exploitation on already-explored good nodes. Thus, this method pairs better with those policies that try to find best nodes early. As for the contrary policies or just random simulation, this method might offer no better result.

The Early Cutoffs method adapts the GOAL definition of GDL. By using this term to evaluate the choices, the method will be able to choose more stable features when running simulation phase in MCTS. The repeating process as stated above is another target of cutoff algorithm. For the stability evaluation, the researchers take the first state when the goal changes and the number of players into account. The first state when the goal changes suggests that the feature has reached its terminal, further actions will be irrelevant choices. If this simulation steps exceeds this value, then the simulation will be terminated and a reward of draw will be given. Similarly, for the repeating terminals, a similar cutoff depth will be given to decide where the simulation has terminated.

Even though these two methods are effective improvement method to the MCTS/UCT when applying to GGP agents. However, these techniques still are arbitrary methods considering the sharp cutting. They will behave better cooperating more aggressive policies which try to run more iterations for more learning data.

Considering that domain knowledge can strengthen UCT performance in a very effective way, many researchers try to generate features learned by algorithm to guide

UCT in simulations. This approach can be generally named GUCT (Guided UCT) [1]. Some researchers have worked on this idea and applied them in the GGP agents designing.

This first MCTS-based GGP agent was CadiaPlayer [15]. The original incarnation of CadiaPlayer used a form of history heuristic and parallelization. The researchers of CadiaPlayer point out that the suitability of UCT for GGP random simulation can implicitly capture the features of a game that heuristic evaluation function have difficulty dealing with. They used their enhanced UCT to build CadiaPlayer can participated in 2009 & 2010 AAAI GGP competition. Though they didn't win the game that year, a general increase can be noted afterwards.

They combined the UCT with several enhancements, including Move-Average Sampling Techniques(MAST), Tree-Only MAST, Predicate-Average Sampling Technique and Rapid Action Value Estimation(RAVE). They found that each enhancement improved performance in some games, while the combination of different enhancements didn't prove to be superior. Here we briefly describe one of these enhancements, RAVE:

RAVE: "Rapid Action Value Estimation" is a popular "All Move As First" (AMAF) enhancement in Go Programs such as MoGo [1] [16]. First we need to introduce AMAF. AMAF's idea is to update statistics for all actions selected during a simulation as if they were first action applied. This algorithm treats all moves played during selection and simulation as previous selections step, meaning that reward of every step is updated whenever and wherever they are encountered. AMAF is a substitute of UCT but practically it is combined with UCT. RAVE blends these two evaluations denoting a UCT score and an AMAF score. The two scores are linearly interpolated using parameter α. This α is given by a function of n visits to a node. And when proper k simulations have been operated, this algorithm will automatically become UCT alone.

Respecting their achievement, researchers from Poland proposed further modification on Guided UCT, their approach was to generate features using generalized GDL expressions. For instance, they generalize features like (cell 1 ? b) to get sub expressions (cell 1 ? b) (cell 1 ? ?) (cell ? ? b) (cell ? ? ?). And then specialize

them to get features like (cell 1 1 b). After operating on the features, they will evaluate them based on their stability using function of Stability=TV/(TV+10SV), where TV denotes total variance of the feature and SV sequence average variance.

4. Game Description Language for incomplete information games

The original version of GDL has a fundamental limitation, which is its restriction to a deterministic game with complete information about game states. Agents can make decisions taken all the information in to account and no hidden moves are included. For games like Bridge, Texas Hold'em, the decision is based on the prediction of opponents' holding cards and the chances of public cards. MCTS is capable of operating random variables, considering the guesses in these games. But GDL doesn't include expressions of such games. So, Michael Thielscher proposed some expansion on GDL for incomplete information games [5]. Incomplete games are quite frequently seen in games. This is one of the game mechanics to give players fun experience and somehow increase the length of the game playing when given limited information for making choices. In tabletop games, dices and card docks are normal, which encourages some special strategies. Some of the strategies are not practical for computer to utilize. However, predicting features are practical for AI agents to apply. And GDL-II, which has RANDOM and SEE expression are capable of describe such behavior for agents.

Basically, term RANDOM refer to the randomness generator (e.g. In a Texas Hole'em, the public cards dock can be considered as a player who randomly distribute cards to other players). Such behaviors include dice rolling, random piece placement, card shuffling. Based on this virtual player's behavior, other players can still be "rational" and make concrete choices that can be described. Another term is SEE. This term denotes the content actual players (i.e. agents) can confirm in the game. The predictions are based on the returned value of this term. The keywords of GDL-II are in the following figure, which is an expansion of GDL [5]:

role(R)	R is a player
init(F)	F holds in the initial position
true(F)	F holds in the current position
legal(R,M)	R can do move M in the current position
does(R,M)	player R does move M
next(F)	F holds in the next position
terminal	the current position is terminal
goal(R,N)	R gets N points in the current position
sees(R,P)	R perceives P in the next position
random	the random player

The detailed definition of GDL-II is much more complex, like GDL's concept, these definitions make game representations flexible due to their concept-level statements. The researchers consider GDL-II as top-down concepts, the comparing language is Gala, which is based on the actual execution tree. The differences are quoted here for comparing and enlightening [1]:

(1) Gala requires an explicit enumeration of the sequence of steps from the beginning to the end of a game;

(2) A Gala description is understood by an execution tree, which corresponds to a game tree in so-called extensive form—hence, the semantics of Gala is procedural rather than declarative;

(3) Understanding a Gala "program" requires to build the entire execution tree for this program, which is unsuitable as a basis for GGP research where the focus is on learning how to play well in games that cannot be searched completely.

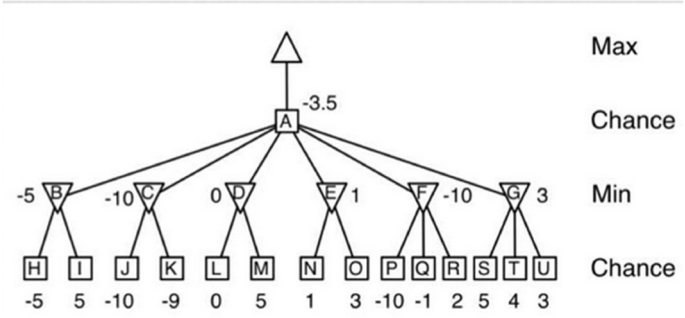
4. Monte-Carlo Tree Search in a Modern Board Game Framework [10]

Tabletop games (Modern Board Games) AI designing is a good topic for game research. As game industry develops, more games are developed with non-determinism and imperfect information to increase complexity and fun. Normally it's not easy for AI to operate on the chance factors in these games. Both the complexity and unpredictable results will add more difficulty to decision making. Another problem is that for different games, there are different rules that matters regarding the theme of the games. Some games encourage aggressive strategies while some encourage

conservative decisions. Thus, it is hard to abstract the game into a model without supervising domain knowledge.

As a statistical method, MCTS can be more efficient than rule-based heuristic methods when implementing a framework for different games. Then the problem is how to describe the games and translate the process into programmable descriptions. In this research, G.J.B.Roelofs suggested a framework, which is a state machine supported by an event-driven graph structure to represent the board of the games. The state machine is composed of multiple State Cycles, each containing Game States that represents the phases of the turn of a player. And then the framework has two other terms, Actions and Game Logic. These two terms are used for deconstruct the games into sub descriptions. This framework suggests some practical implementations on how to interpret the game mechanism of tabletop games. In this research, the game Settlers of Catan was chosen as a proof-of-concept implementation.

Another implementation of the framework is chance node in MCTS, which is used to operate on non-determined components of the game. A chance node is integrated in to MCTS in stochastic process as a EXPECTMAX term, as shown here:



This research is helpful on how to interpret the game mechanism into generalized form. To design algorithms that can play different games, how to abstract the game model will have crucial influence. And the idea of combining EXPECTMAX term might also suggest ways on how to integrate different components in decision making.

III. System Modeling and Structure

1. Game Abstraction

To implement desired algorithm on games in this study work, it is necessary to abstract the game resource system into a generalized form. In many case, a generalized model requires sufficient simplified rules and description. Considering that the objective of a resource management AI is to maximize the value of functional units with sufficient resource generating units as support. Here I want to briefly describe these terms first.

In many computer games, the playing patterns can be abstracted into such a description:

Step 1: Deploy Resource Collection Units. (e.g. Farmers in Warcraft, Settlements in Settlers of Catan). This step is the foundation of the game. Normally there is a linear relationship between collection units and resources that player can acquire. But that doesn't mean the winning probability is linear. If the players focus too much on resource collection, there is high chance that they lose in the game, because the collection units are often not functional. Like in Warcraft, the main gameplay is about building the army instead of getting more farmers.

Step 2: Acquire Resource in consecutive iterations. This step is to accumulate resource and at mean time, allowing for more units to be deployed. This is a parallel process with deploying functional units. As it's the prerequisite of deploying units, it has higher priority in actual game process.

Step 3: Deploy Functional Units. The functional units refer to the units which take effect in the gameplay and determine the result of the game. Though players' controlling skill is also an important factor to decide the result of the game, it is not considered in this study. Another specification to mention is that the functions are neutralized here as integer values to simplify the system construction. That's because different games have different sets of functions. And more importantly, these functions' relation is likely to resemble Morra game due to the balance designing purpose. In actual cases, this abstraction is probably incorrect because the game mechanisms are not linear functions, but with tradeoff and other factors. But broadly speaking, it can represent a value function based on domain knowledge of the game.

Step 4: Evaluate All the Units. As stated above, the units might have complex

functions regarding to specified games. Since that such analysis is largely based on the fundamental rule of the game, we may consider this part of calculation is not able to be generalized. Thus, here a simple sum is used as evaluation method.

Except for this basic game cycle. Another important factor to mention is the temporal value. There is a popular logic in many games: more powerful units require longer time to deploy. The tradeoff between temporal value and functional value is critical in these games. For traditional board game, this might be an invisible to the players, like how to evaluate the whole board of a GO game. Normally the temporal value can be both discrete or continuous. For tabletop games, it's probably the turn, which is discrete value. For real-time strategy games or some simulation games, this value is continuous. Here we name the temporal value as *Iteration*. Then the game process can be described as *Iterations* of the 4 steps mentioned above.

To build a tree with clear structure, it is better to use discrete *Iterations*, in other words, a turn-based game simulating model. Thus, we need to interpret the real-time games into a turn-based form. A simple method is to divide the time domain into separated small fixed intervals, like 10 secs or 1 min. The resource acquired in each turn is determined by the collection units. And the amount is fixed in one turn. Practically, this simplification might be wrong because the resource collection is also a continuous process in real-time game. Here we assume the change of collection units is relatively small in a single turn. Another hack to simplify the model is to set a max building number for each unit in one iteration, indicating the temporal cost of the units. This is mainly for those powerful units which require more time to build. Because here a turn-based process is used for the model, it will be hard to simulate the sequential unit deploying process like in many real-time strategy games. (e.g. In Warcraft, the units are deployed one after another, each takes a fixed amount of time.)

In practical view, abstracted game model can be implemented by 4 modules: Unit Module and Server Module, AI Module and Adversary Module (Optional):

a. Unit Module: The Unit Module is used to assign attribute to the units in the game simulation. The Attributes of the units are: *Cost*, *Value*, *Speedup, Max-build*. *Cost* refers to the resources needed to deploy one unit of the kind. *Value*

denotes how much value the unit worth, which represents the unit's function effectiveness in an actual game. The *Speedup* refers to how much more resource can be generated by the unit in each iteration. The *Value* attribute is mainly for functional units, the *Speedup* Attribute is mainly for collection units

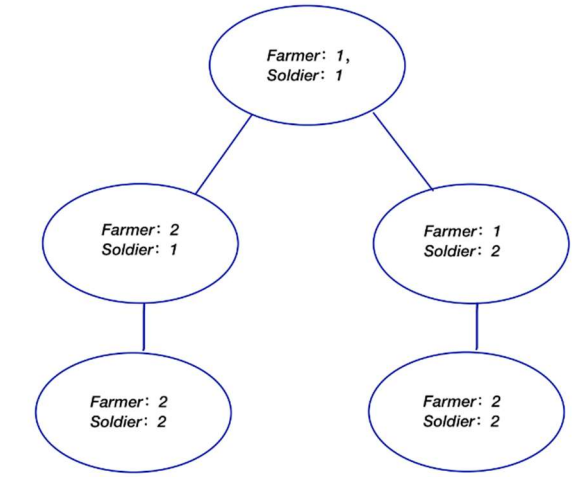
b. Server Module: The Server Module is used to control the whole process of the game simulation. In this module, the acquired and consumed resource amount is updated in each iteration, based on the units' status. Meanwhile, this module will evaluate the total value of the units. Another important function is that it will operate on the *orders* from players (i.e. AI or human). In this case, the *order* means what units to deploy in a single turn by the player.
c. AI Module: AI Module is used to implement MCTS tree search algorithm. It will receive the output from Server Module. After calculation, it will return an order to the Server Module. The status evaluation will all be done in the Server Module. The AI Module's function is basically about how to decide the best deploying plan in the next round.
d. Adversary Module (Optional): This module is used to receive human's input as the adversary to AI. It might be possible to replace this module by another AI module with rule-based heuristic algorithm. Or a naïve policy can be implemented in the MCTS AI and this module can be omitted.

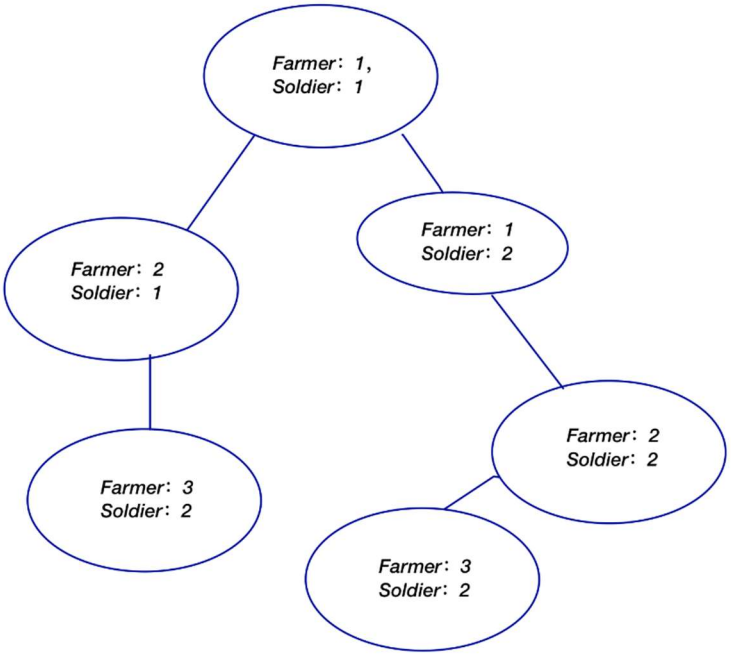
MCTS Specifications

To construct a search tree, we need to consider how to specify the node attribute. Under UCT algorithm, the most popular MCTS algorithm, each node has a simulation count and a winning count, referring to how many times this node is visited in all the simulations, and how many time a winning result is reached starting from this node. In most cases, the simulation is a stochastic process.

In this study work, the tree node's key can be regarded as a form like ({Unit1: N1, Unit2: N2, Unit3: N3……}, *Iteration*). The first attribute is a dictionary referring to current units' status of the player, including pointer of each unit and their corresponding number, denoted as $Unit_i$ and N_i. The second value is the temporal value,

Iteration, i.e. round of the game. The reason to implement this structure is that it is possible to reach the same status of units at different (or the same) iterations, as shown in the following figure:





The reason why such situation might occur is that there are chances that some resource gap exists. Like in the second graph, on the left path, the player chooses to deploy farmer earlier, thus allowing 1 farmer and 1 soldier to deploy in the next iteration. In comparison, he right path is slower. Such a situation is frequent in strategy games. The two leaf nodes in the previous examples are the same status. And the visit times of these nodes are distributed because they have different parent nodes. This problem can be optimized using AMAF (All Move At First) algorithm, which means all

the moves are considered as a potential move.

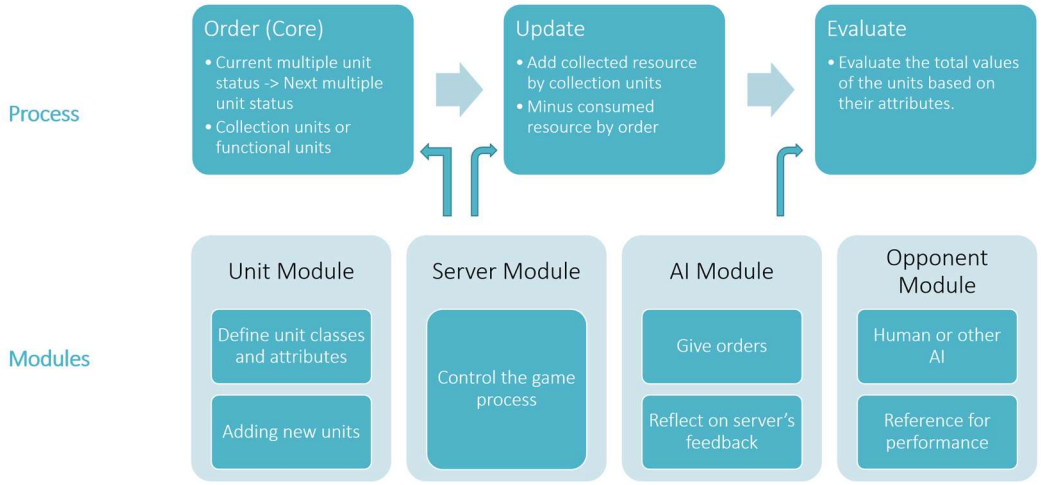
The detailed construction of the tree search process will be discussed in the following part of the report.

IV. Methodology and Algorithm in implementation

Basically, the methodology is to associate different attributes for evaluation and make the decision that is balanced between exploration and exploitation of the units' deployment in real-time strategy games and tabletop games. To implement this algorithm, each attribute will have a corresponding search tree for the association analysis. The analyzing method is an expanded form of UCT algorithm. And game AI will finally return a move in the game given an initial status.

1. Basic Description

For each attribute in the decision reference list, an isolated tree is built. And during the Selection step in MCTS process, the trees will be searched downwards simultaneously. The searched nodes move towards the nodes whose combination can maximize the associated UCT value. And during the back-propagation step, the node visit times and winning times will be updated simultaneously for each attribute.



2. Details of MCTS

MCTS's core description is a 4-step loop: [Selection, Expansion, Simulation, Back-Propagation]. The process can be implemented as an iterated process:

```
Function Selection () {
    Selection Process;
    If Time Out:
        Return Result
    Else:
        If Need-Expand:
            Expansion ()
        Else:
            Selection (deeper level)
}
Function Expansion () {
    Expand Process;
    Simulation Process;
    Back Propagation Process;
}
```

In this study work, there are multiple trees to operate in a single searching iteration. Thus, a dictionary structure is applicable in implementation. The basic data structure to store the data is a 3-level dictionary: {Unit Ref: {Game Round: {Unit Increase Number: Visit Times}}}, {Unit Ref: {Game Round: {Increase Number: Win Times}}}.

1) The first level "Unit" is used to locate the corresponding tree for a specific unit attribute (e.g. A tree for the farmer units). And this is also the mostly frequently used reference in the program because the tree-related operation need to be done to proper tree.
2) The second level is the "Game Round", which represents the "*Iteration"* described in the previous part. This value is used to proceed the game status updates. And in MCTS, this value is essential in the selection and simulation process because we need to use it for monitoring the descending search. In other words, this is the vertical cursor for the nodes.

3) The third level is the “Unit Increase Number” value. This value is the horizontal cursor for the nodes. It represents the existing nodes in the same level. In the UCT algorithm, this value is used to get the visit time of node’s parent node.

In fact, this data structure is not a tree. It is a graph whose nodes are placed in a two-dimensional network. This algorithm is like the AMAF algorithm, which is another variation of MCTS. This method treat all the nodes as available nodes. Even so, the searching process should still be a tree-like behavior.

Practically, the visits count dictionary can be merged with wining count dictionary. But separating can simplify programming.

3. Associated UCT function

The original form of UCT function is

$$\frac{w_i}{n_i} + \ c_1\sqrt{\frac{\ln t_i}{n_i}}$$

Where w_i denotes the winning times of the node, n_i denotes the visit times of the node. And t_i denotes the parent node’s visit times (I.e. sum of visit times of siblings and the nodes itself). This function is good for balancing exploration and exploitation of the nodes selection. Suppose we need to associate different attributes to make decision now. An intuitive way is to use the combination of the attributes as a node. And generate possible nodes for the following process. However, if the attributes have a larger number of possible values. The combinations’ counts will grow exponentially, which requires a lot of memories to store the information. And meanwhile, the normal MCTS only does expansion one time in a single loop. By using isolated tree, it is possible to save storage, and allowing more possible expansion.

An intuitive expansion of the UCT function is using

$$\sum_k \frac{w_{ik}}{n_{ik}} + \ c_{1n}\sqrt{\frac{\ln t_{ik}}{n_{ik}}}$$

Here n denotes the number of the trees. To maximize this value, we just need to extract the node that maximize UCT value from each single tree. The is because the trees separated. The problem is that: for such an equation, the value will be biased

separately to each single tree's maximum balanced search process.

However, if we want to take the interconnection of these units into consideration, this function will not help much. If one attribute has a smaller number of possible values, the tree will have also have less nodes. Considering that the trees grow simultaneously, the corresponding update count of visit times and winning times will be distributed evenly. Thus, for this attribute, the visit times and visiting times will be larger because it has an inverse relationship with number of nodes. In another word, for such an attribute, they can potentially be more powerful items due to the game design philosophy. In the above function, the UCT value for one attribute is calculated by the ratio. Thus, there probably is no dramatic difference between different UCT values. To allow powerful units have larger weight in decision, we may consider merge the visit times and winning times.

Here is a method that merge the exploitation term, named equation (1):

$$k\frac{\sum w_{ik}}{\sum n_{ik}} + \sum_{k} c_{1k}\sqrt{\frac{\ln t_{ik}}{n_{ik}}}$$

And equation (2), merge the exploration term:

$$\sum \frac{w_{ik}}{n_{ik}} + kc_{1k}\sqrt{\frac{\ln \sum t_{ik}}{\sum n_{ik}}}$$

And equation (3) merge both terms:

$$\frac{\sum w_{ik}}{\sum n_{ik}} + kc_{1k}\sqrt{\frac{\ln \sum t_{ik}}{\sum n_{ik}}}$$

Here in this study, (1) is applied, which means let exploitation factor affect the decision more.

4. Implementation Problems and Methodology

1) Expansion Choice

In the expansion step, a new node is chosen to be added to the tree. A simple way is to select a random node from available nodes. Basically, how to choose the node will not affect the result. But MCTS normally lead to an asymmetric structure search tree after converges. With proper choosing algorithm, the

process can be accelerated.

Considering that in most games, the mechanism doesn't encourage players to hold the resource. And players also want to make more choices in the games. Under such an assumption, it might be better for the AI to choose larger deployment number.

Thus, a possible solution is to expand a maximized legal node to do the simulation. And after that, the expansion node can move towards the choices with less deployment number

2) Dynamic Resource

In the computer game resource system, the resource normally doesn't accumulate in a linear way. And at meantime, the resource is also consumed from time to time. Thus, the resource is in a fluctuant pattern. In this implementation, the nodes are visited as a combination. Because resource and combination might both change during the selection process, the nodes are also not visited in a determined pattern.

To solve this problem, we may use the Iteration to update the status. And associate it with the number of units to increase to represent the node. As stated above, this method will change the data structure to a graph. Also in the server module, the update function is essential.

3) Legal Test

Because in this implantation, the combination is also a variable. Thus, there are possible cases that the combination created using existing nodes exceeds the resource limit even that the nodes are all visited and considered to be available for selection.

Thus, we may implement a legal test in the selection step. If the combination is not legal given the status, then this combination will be omitted. And next maximum UCT combination will be tested until a legal combination is found. It is also applicable if we choose a brute-force decreasing, like minus the largest cost unit count by one. But such an implementation is arbitrary and it is probably not the largest UCT value combination.

4) Independent Exhaustion

Because the trees are expanded separately. And each tree might have different number of nodes to choose from in the expansion step. There are chances that when a tree needs expansion, the other's nodes at this iteration is already exhausted and doesn't need to be expanded.

A possible and simple solution is not to expand simultaneously. Given that one tree need expanding and another exhausted, the exhausted tree will not expand. The whole tree only grow deeper when all the trees are exhausted in current iteration level.

V. Preliminary Performance Analysis

The performance of the algorithm can be analyzed through a comparison with original UCT algorithm, which is using the combination as the node of the search tree. The analysis is based on preliminary computation complexity and example case.

1. Testing Base Description:

The test base is a simplified real-time strategy game. There are 3 available units in this game: Farmer, Soldier, Hero. Their attributes are:

{Farmer: cost = 6, value = 3, speedup = 3, max-build = 20}

{Soldier: cost = 10, value = 10, speedup = 0, max-build = 50}

{Hero: cost = 100, value = 150, speedup = 0, max-build = 3)

The mechanism is basically described as:

Player can choose units to deploy in each round. After player give the order of deployment, the resource will minus the cost of the order. Then the status will be updated. Units' number will increase based on the description of the order. Then next iteration begin and player's resource will increase by the number of acquired collection units multiply its speedup value.

The unit "farmer" is used to represent the resource collection units. The unit "hero" is to reflect the exhaustion behavior of AI agents. And "soldier" reflects the leverage

of total value. And there are four attribute for each unit: cost, value, speedup, max-build. Cost is the resource consumption for deploying one unit of this kind. Value is the evaluation reference, which often represent the battle power in strategy games or other functional value. Speedup is the dynamic bonus resource per round that base on the number of units who can collect resource. Max-build is the max number of units that one unit can build per round.

The game server status is initialized as:

{Init-Resource: 30/40/50, Add-up: 10, Unit Number: (0 farmer, 0 soldier, 0 heroes)}

Init-Resource is the resource amount each player has at the beginning of the game. Add-up is the constant resource bonus per round, which can enable AI have more choices.

Considering a large enough number of steps is applied, the final status will converge to exhaustion deployment because the resource will be redundant. Thus, we may assume that a better performance is achieved with less moves in the beginning to obtain sufficient resource and units. However, in actual games, the units can also be consumed, like starting a battle or making a trade. Thus, here the comparison will be mostly on how to achieve higher total value in a few rounds.

2. Testing Result:

Testing is to compare four different modified UCT functions referred in above part of report. Due to reasons of program optimization, the test method is confined in a *ten-round* game process. And each complete operation is a set of ten games. The maximum MCTS search times is 100. Since tree depth is a dynamic variable, it will be automatic generated. Also, the initial resource will also affect the result. Thus, the result is shown in three columns by initial resources. The comparative is a random playing opponent. Because this model is an isolated gaming process, independent testing can be applicable.

Playing process example (to interpret how the game works):

In each cell, the presented values are the units' number and resource at the ending of this round. And the sample testing's initial resource is 30

AI	Random Play
soldier: 1 farmer: 2 hero: 0 Resource Status: 24	soldier: 0 farmer: 1 hero: 0 Resource Status: 37
soldier: 2 farmer: 4 hero: 0 Resource Status: 24	soldier: 0 farmer: 2 hero: 0 Resource Status: 47
soldier: 3 farmer: 6 hero: 0 Resource Status: 30	soldier: 2 farmer: 3 hero: 0 Resource Status: 40
soldier: 4 farmer: 8 hero: 0 Resource Status: 42	soldier: 2 farmer: 5 hero: 0 Resource Status: 53
soldier: 6 farmer: 10 hero: 0 Resource Status: 50	soldier: 4 farmer: 8 hero: 0 Resource Status: 49
soldier: 9 farmer: 13 hero: 0 Resource Status: 51	soldier: 4 farmer: 9 hero: 0 Resource Status: 80
soldier: 13 farmer: 14 hero: 0 Resource Status: 57	soldier: 6 farmer: 18 hero: 0 Resource Status: 70
soldier: 16 farmer: 18 hero: 0 Resource Status: 67	soldier: 10 farmer: 18 hero: 0 Resource Status: 94
soldier: 22 farmer: 19 hero: 0 Resource Status: 68	soldier: 18 farmer: 19 hero: 0 Resource Status: 75
soldier: 23 farmer: 28 hero: 0 Resource Status: 98	soldier: 22 farmer: 22 hero: 0 Resource Status: 93

Note that this game model is a single player game, here the comparison is just to have a better reference to illustrate the updating process.

1) Function 1: $k\frac{\sum w_{ik}}{\sum n_{ik}} + \sum_k c_{1k}\sqrt{\frac{\ln\ _{ik}}{n_{ik}}}$

cell: (Win, AI Score, Random Score)

Initial Resource = 30			Initial Resource = 40			Initial Resource = 30		
Win	AI Score	Random Score	Win	AI Score	Random Score	Win	AI Score	Random Score
0	302	336	0	288	325	0	349	441

1	232	245	1	260	203	1	442	205
1	278	149	1	259	169	1	267	212
1	329	321	1	600	260	0	299	318
1	259	144	1	259	225	0	187	274
1	282	250	1	289	217	1	518	513
0	282	312	0	292	298	1	373	335
0	278	356	0	227	430	1	617	413
0	250	262	1	289	234	1	657	292
1	278	181	1	278	365	0	473	529
Winning Rate: 0.6			Winning Rate: 0.7			Winning Rate: 0.6		

2) Function 2: $\sum_k \frac{w_{ik}}{n_{ik}} + \sum_k c_{1k}\sqrt{\frac{\ln \ _{ik}}{n_{ik}}}$

Initial Resource = 30			Initial Resource = 40			Initial Resource = 30		
Win	AI Score	Random Score	Win	AI Score	Random Score	Win	AI Score	Random Score
1	406	365	1	326	310	1	773	262
1	320	166	0	227	283	0	212	299
0	335	464	1	402	252	0	212	359
1	314	299	1	316	130	1	730	311
1	316	289	1	471	211	1	419	287
1	277	200	1	603	307	1	499	383
1	320	233	1	412	431	1	773	243
1	292	235	1	252	135	0	252	385
0	316	347	1	237	203	1	459	232
1	368	225	1	354	209	0	193	253
Winning Rate: 0.7			Winning Rate: 0.9			Winning Rate: 0.6		

3) Function 3: $\frac{\sum w_{ik}}{\sum n_{ik}} + c\sqrt{\frac{\ln \sum t_{ik}}{\sum n_{ik}}}$

Initial Resource = 30			Initial Resource = 40			Initial Resource = 30		
Win	AI Score	Random Score	Win	AI Score	Random Score	Win	AI Score	Random Score
1	368	309	1	720	402	1	831	362
0	349	372	0	336	424	0	200	316
1	341	218	0	343	350	1	831	279
1	349	245	1	391	201	0	273	333
0	349	355	1	381	236	0	200	391
1	341	236	1	340	308	1	661	529
1	424	177	0	266	328	1	831	440
1	349	279	0	266	359	0	332	357
1	266	212	0	330	356	1	290	288
1	355	312	1	359	331	1	615	427
Winning Rate: 0.8			Winning Rate: 0.8			Winning Rate: 0.6		

4) Function 4: $\sum_k \frac{w_{ik}}{n_{ik}} + c\sqrt{\frac{\ln \sum t_{ik}}{\sum n_{ik}}}$

Initial Resource = 30			Initial Resource = 40			Initial Resource = 30		
Win	AI Score	Random Score	Win	AI Score	Random Score	Win	AI Score	Random Score
0	214	236	1	581	181	1	239	208
0	219	372	1	279	189	0	197	303
1	571	265	1	581	190	0	213	345
0	214	282	0	219	259	0	259	277
0	239	277	1	730	178	1	740	385
1	571	228	0	207	251	0	197	258
1	571	268	1	730	256	1	740	283
0	219	245	1	581	324	0	217	438

0	239	367	1	229	175	1	591	407
0	239	290	1	249	218	0	217	326
Winning Rate: 0.3			Winning Rate: 0.8			Winning Rate: 0.4		

Winning Rate Sheet (Each cell represents a winning rate of 10 games):

Function,	Initial Resource	1st	2nd	3rd	4th	5th	6th	7th	8th	9th	10th
Function1	30	0.6	0.7	0.7	0.5	0.9	0.6	0.6	0.7	0.6	0.7
	40	0.7	0.4	0.5	0.7	0.8	0.7	0.6	0.5	0.7	0.8
	50	0.7	0.6	0.3	0.4	0.7	0.8	0.5	0.3	0.7	0.6
Function2	30	0.8	0.6	0.7	0.9	0.6	0.7	0.6	0.8	0.8	0.6
	40	0.8	0.6	0.7	0.5	0.9	0.9	0.7	0.8	0.7	0.9
	50	0.5	0.5	0.4	0.7	0.6	0.6	0.4	0.3	0.6	0.5
Function3	30	0.8	0.9	0.8	1.0	1.0	0.7	0.9	1.0	1.0	0.9
	40	1.0	0.8	1.0	1.0	0.9	1.0	0.9	0.8	0.6	1.0
	50	0.8	0.9	0.7	0.9	0.7	0.5	0.9	0.7	0.8	0.7
Function4	30	0.5	0.4	0.5	0.3	0.2	0.3	0.7	0.4	0.3	0.2
	40	0.5	0.6	0.7	0.5	0.4	0.6	0.8	0.7	0.8	0.4
	50	0.2	0.3	0.3	0.2	0.3	0.4	0.2	0.5	0.2	0.4

Average (left) & Variance (right)

Initial Resource	30		40		50	
Function1	0.66	0.0104	0.64	0.164	0.56	0.284
Function2	0.71	0.109	0.75	0.165	0.51	0.129
Function3	0.9	0.01	0.9	0.016	0.76	0.144
Function4	0.38	0.0216	0.6	0.02	0.3	0.01

3. Result Analysis

Due to programming and hardware issue, the program performance is limited.

Only a small amount of testing can be done each time. And the gaming results are also posted here for analyzing possible issues. The first test is gameplay results and the second is on winning rate. Because this game model has self-feedback, as the process goes on, the outcome may vary dramatically. The score can reflect how the game is biased due to specific implementation.

Here is some possible analysis from the result:

1) The MCTS algorithm will return the same results for many times. This is probably because the game model is simple enough. Also, the values are discrete choices. Thus, in few steps, the choices might be largely biased to some specific choices and the results will converge to some values.
2) The score is sensitive to the initial resources somehow. And for larger initial resource, the winning rate will be more unstable. In other words, the performance will decrease. There are three possible reasons, program implementation, game model or algorithm itself. In view of grogram implementation, the reason might be the garbage collection or other performance problem. As for the game model, the possible case is that the game's complexity increase dramatically. And algorithm design has a similar effect on the result. Further analysis is required for concluding.
3) Approximately, function3 has the best performance among four modified forms, followed by function2 and function1. Function 4 has the worst performance. Overall speaking, the performance will decrease as initial resource increases.

VI. Conclusion

This study work goes through some popular research related to Monte-Carlo Tree Search Algorithm, and its related usage in General Game Playing. Considering that many modern computer games have similar patterns of resource management. It would be useful to implement a resource management algorithm that can work for different computer games given necessary input based on MCTS

Then the study is on how to abstract the game model and describe the resource management process using several attributes. Normally it is possible to describe the resource management with 4 actions: Deploy Resource Collection Units, Acquire Resource in consecutive iterations, Deploy Functional Units, Evaluate All the Units. Thus. And some simplification assumptions are required for further implementation. The first simplification is using finite time range to format real-time process into discrete round-based process. With this step, it is possible to assign nodes to the tree. Then, the node values are integers representing the visit times and winning times in MCTS. The resource process is also discrete. For each iteration, the resource will be updated two times: the first is accumulation based on initial status, the second is consuming resource to deploy some units. Another reason to format the process is that the deployment time of one unit can be transformed into the maximum legal deploy number in a finite time interval.

To implement this algorithm, a module structure is proposed. The program has 3 main modules and 1 optional module. The main modules are unit module, server module, AI module (using MCTS), the optional module is the adversary module which is controlled by human or other AI agents. On MCTS sides, the tree node keys are specified by the iteration and adding count. In other words, the search tree represents orders to the game server.

With these prerequisites, the algorithm implementation is discussed in this study work. The basic idea of the algorithm is to build a search tree for every kind of units. Then conduct an associated selection with the modified UCT algorithm. This method can be considered as a modified tree policy. The purpose to suggest such an algorithm is to ease the storage space and to allow more possible combination of units.

Then the theoretical performance analysis is discussed in the final part of the study work. Four different modified UCT functions are applied to test the performance. And testing shows that the AI's result will converge to some values due to the biased choices. And for different initial resource amount, the performance will change. Smaller initial resource tends to have a better performance. And among the four different functions, function3 has the best playing results.

Overall, this study work suggests a modification on UCT algorithm that might be useful when designing the resource management AI module. And there are some flaws that need to be solved. The first problem is the data structure. In fact, this search tree is a graph. And another concern is the comparing reference. Here no other AI or human behavior is tested in comparison to the associated UCT function. These are the potential future work that can be done. One typical comparison is the basic UCT function. Given enough storage and memory. The UCT tree of game orders can be applied. In this case, each node will represent a determined behavior in the game. There might be redundancy because basic UCT algorithm has no AMAF value. Another proper testing is to compare the AI performance with human input as comparison. Since the game is an individual and complete information game. Most optimized deployment plan is indeed applicable. Overall speaking, these are the possible work for the future study.

VII. Reference

[1] Browne C B, Powley E, Whitehouse D, et al. A survey of monte carlo tree search methods[J]. IEEE Transactions on Computational Intelligence and AI in games, 2012, 4(1): 1-43.

[2] Walędzik K, Mańdziuk J. Multigame playing by means of UCT enhanced with automatically generated evaluation functions[C]//International Conference on Artificial General Intelligence. Springer Berlin Heidelberg, 2011: 327-332.

[3] Finnsson H. Generalized Monte-Carlo Tree Search Extensions for General Game Playing[C]//AAAI. 2012.

[4] Thielscher M. A General Game Description Language for Incomplete Information Games[C]//AAAI. 2010, 10: 994-999.

[5] Genesereth M, Love N, Pell B. General game playing: Overview of the AAAI competition[J]. AI magazine, 2005, 26(2): 62.

[6] Finnsson H, Björnsson Y. Game-tree properties and MCTS performance[C]//IJCAI. 2011, 11: 23-30.

[7] Méhat J, Cazenave T. Combining UCT and nested Monte Carlo search for single-player general

game playing[J]. IEEE Transactions on Computational Intelligence and AI in Games, 2010, 2(4): 271-277.

[8] A Python N-in-Row game based on Monte Carlo Tree Search and UCT RAVE https://github.com/arrti/mcts/blob/master/n_in_row_uct_rave.py

[9] Roelofs G. Monte carlo tree search in a modern board game framework[J]. Research paper available at umimaas. nl, 2012.

[10] The General Video Game AI Competition. http://gvgai.net/index.php

[11] Pell B. METAGAME: A new challenge for games and learning[J]. 1992.

[12] Love N, Hinrichs T, Haley D, et al. General game playing: Game description language specification[J]. 2008.

[13] Szita I, Chaslot G, Spronck P. Monte-carlo tree search in settlers of catan[C]//Advances in Computer Games. Springer Berlin Heidelberg, 2009: 21-32.

[14] Finnsson H. Cadia-player: A general game playing agent[J]. 2007.

[15] Gelly S, Silver D. Combining online and offline knowledge in UCT[C]//Proceedings of the 24th international conference on Machine learning. ACM, 2007: 273-280.