

Abstract:

What will you feel when play with an unchangeable AI in RTS game? Of course, it is boring. You can beat them easily and that's no fun. In this research, we will try to design an AI with learning-ability and return the fun to players. We firstly abstract a simple AI mode, and then implement a suitable learning method. Due to some developing problems, we simulate the system (ORTS). Finally, we establish a new learning system for RTS AI. It's an advanced point system based on the conception of the evaluation system in commercial RTS game. Decision making processes could depend on the points of each unit. Point is calculated by unit information, current game states and "experience" and. The increase "experiences" lead the value to a precise number. These changes would affect some process and up to the whole game.

Table of contents

Abstract:	1
1. Introduction	4
1.1 Outline	5
2. Background and relevant literature	7
2.1 RTS game	7
2.1.1 Units.	8
2.1.2 Technology tree	9
2.1.3 Terrain definition.	10
2.2 ORTS	11
2.3 AI in RTS	12
2.4 Relevant learning theories	14
2.4.1 Learning automaton [7]	14
2.4.2 Decision tree	17
2.5 Other related conceptions	19
3. Problem description and analysis	21
3.1 Sub problem.....	21
3.2 Simple solution to sub problems	22
3.2.1 Type of RTS game [12]	22
3.2.2 ORTS develop environment	24
3.2.3 Rush-attack.	26
3.2.4 Learning method	28
3.3 Requirement specification	28
3.3.1 Experiment goal	28
3.3.2 Game Specifications	28
3.3.3 Strategy specification.....	30
3.3.4 Learning specification	30
4. Design	31
4.1 AI blueprint.....	31
4.1.1 Three-layer architecture	31
4.1.2 Our design.....	32
4.2 Learning theory	37
4.2.1 Main learning method.	38
4.2.2 Decision making process.	41
4.2.3 Sum up.....	46
4.3 working with ORTS	47
4.3.1 Problem 1: lower FPS	47
4.3.2 Problem 2: move continuity	48

4.3.3 Problem 3: building	49
5. Implementation	50
5.1 System overview.....	50
5.1.1 Overview	50
5.1.2 Map in three-layer architecture.....	51
5.2 Implementation of each class.....	52
5.2.1 Worker	52
5.2.2 A_object.....	53
5.2.3 B_object.....	54
5.2.4 Gui.....	55
5.2.5 Alcore.....	56
6. Testing.....	59
6.1 Correctness	59
6.2 Efficient	61
6.3 Reactivity.....	63
7. Discussion	65
8. Conclusion.....	67
REFERENCES	68
APPENDIX	69
A1 SEARCH TABLE	69
A2 CODE RESOURCE.....	70

1. Introduction

Commercial computer games is an important parts of entertainment industry, it is very popular both in young and old people. As we know there are many kinds of games. Simulations based game is the most important one; even it is used as a critical aspect of specialist training, like pilot. By the highly development of personal computers, this kinds of games became very popular. These games always have an environment simulates the real time world. Players are able to control the objects or units in this world. It brings you a real sense while playing. In order to enhance the sense and reactivity of the world, a strong demand of real-time AI this is able to solve real-world decision tanks correctly and quickly. In our research, we concern one of simulation based game: RTS game. We are doing an AI research based on these games.

RTS is short for real time strategy. RTS game simulates a military section. There are some cross-sectional games. Such as million-seller Starcraft and Warcraft produced by Blizzard. In general, several players struggle in a resource scattered map. Player commands the game units to developing economy, climbing technology, crafting new troop. The aim is crafting a powerful troop and guiding them into battle. Afterwards, destroy you opponents' base and troop.

RTS game already developed for over 10 years. There are many good products. In early time, we have command and conquer(C&C) and red alertness. Nowadays, Starcraft, Warcraft are well known in world wide. RTS game is randomly, real time, and unpredictable. Hence, it brings high competition and intensity attention to players, especially when you play with a human player.

However, many players are aware of the AI is so weak and changeless. Why the AI develops didn't follow game step like the relevant arena such as classic board game. There are some reasons:

a) Firstly, **the RTS game world is really complex**. It contains various objects, a huge number of features, partial visible information, micro actions, and real-time decisions. In contrast, current RTS game AI players performance in a n opposite direction. Its turn based, perfect information, most action have a predict sequences. The most important is decision making based on a stable manner.

b) **Commercial RTS game doesn't open its source** for commercial reasons. Also, in order to make the graphics of the game more realistic, most time are spend. It means only a few time is cost in AI research. In addition, lack of resource restrains the third party and fans to develop AI.

c) Most people are interested in human competition. We have a lot competition for humans, such as WCG, ESWC, but **lack of AI competition**. It decreases the

motivation of those people that are able to develop AI. [1]

We'd like to introduce an open source environment: ORTS. It designed for people who are interested in AI research. Also it holds AI competition every year. Our research is based on this environment. We start at the weakness point of currently AI which is the learning ability. In general, the predicted sequence can't handle the situation that would occur in complex RTS world. In this case, a learning ability is a great enhance for AI. It helps AI to learn from the world and change some old fashion and mechanism. However, we can't suppose the AI would do perfectly; it still gives AI more opportunity to make the right decision and win the game. So, we would investigate the learning ability in RTS AI and try to find a suitable one.

In general plan, we divide research process into two steps. At the first, we analyses the standard game information in ORTS. Afterwards, we design our AI package with rush-attack strategy. Our experience of playing RTS game is a great enhance in package design. Then we could investigate learning methods based on it. As the research going, we have deeper understand of RTS game. As a player, we only got unperfected information and unstable game states. It's really hard to find a decision sequences with universal power. Also it is impossible to presents all possible decision sequences. Cause it would be so huge and infinite. Hence, we decide to design a new method. We notice that the evaluation system in commercial games always present the winner with higher point. In same sense, the point gives some clue of the game process. Based on this inspiration, we design an advanced point system. It gives a point to each game unit. AI tries to make decisions to maximal the points. The higher points AI get the higher probability to win. The point is calculated by units' information, current game states and "experience". Experience is a kind of feature that can influence the game in an indirect manner, such as the lost attack power problem. AI learns the "experience" from game to game. It makes the point more precise and game process would change due to the update of "experience". Our implementation uses the lost attack power as the only feature. It gives stronger evidence to the point system.

1.1 Outline

The first chapter of this paper gives an introduction of this AI research project, including some relevant reasons. Then chapter 2 presents background and relevant literatures such as the RTS game, working environment: ORTS, and two learning methods. Afterwards, we re-describe our projects and divide it into four sub problems. The feasible solutions are simply described in the following. At the end of chapter 3, requirement specification is given. In chapter 4, we follow the specification to presents our design in detail. It would divided into three parts, the overview of the AI package, learning theory and the problems we met while implement our design in ORTS. Those insoluble problems described in the end part of chapter 4 lead us to

simulate ORTS environment and implement our design in it . Chapter 5 presents the detail of implementation, including an overview and function details of each class. As always, testing is right behind the implementation in chapter 6. It would test our design in three aspects: correctness, validity and reactivity. Chapter 7 discusses the reasons why we investigate our own learning method instead of use the previous ones. We take the methods described in background as samples. Finally, we come to the conclusion part. It summarizes our work, and presents our design in a nice format. The end part of this paper is references and appendix.

2. Background and relevant literature

RTS AI research covers a large scope. Before the research, we view a large number of related literatures. First of all, we give a well definition of RTS games. Then, a short description of ORTS presents the research environment. The following section defines the AI, and describes some exist AI packages, including the previous AI research conceptions and methods. The last section is a description of previous learning theories. We mention the potential theories that may useful in our research.

2.1 RTS game

RTS is short for real-time strategy. It is an important type of popular game nowadays. It simulates the considerations and circumstances of operational warfare and military tactics. [2] Players struggle in a predicted map, collecting resource, crafting units, climb technology tree, and so on. It's not turn based; player's actions and commands are independently of other players. [3]

RTS games developing over 10 years. There are many good products. We will use one of them to describe the generally game logic of RTS game. We choose the most famous game which prevails 10 years. Its names "Starcraft" produced by Blizzard entertainment.

As most RTS games, player starts with a control center and several basic *units* that used to collection resource. The player is given a top-down perspective of the battlefield. Around control center, the map is invisible. Player need explore the environment himself. There is an abstract action list for each unit, which presents its main function. Player can control them with technique of clicking and drawing. Then player controls these units to build new *buildings* and climb *technology tree* to enable units that is not able to craft or build. Such as, player wants to craft a tank, then player needs to have barracks first, then it enables academy. It requires academy to building factory. Tank is crafted in factories only if factory get its attachment: machine shop. These kinds of restrains in building or crafting draw the outline of technology tree. We would like to presents a partial tree in follow section. Players repeat its actions, like crafting, building to initial his troop. The player which lost all buildings loses the game.

For more detail, we present some important features of Starcraft. It helps to understand how RTS game works. Of cause, features are different from game to game.

2.1.1 Units.

In general, we have two kinds of units in RTS game. Units as basic workers can move around the map, doing some specified work, such as collecting resources, attacking creeps and opponent troop. It called moveable units. The second kinds of units, it is buildings. They can't move. They have function as factories and academy, crafting moveable units, upgrade technology. However, in most satiation buildings can't move, there are exceptions too. For instance, Terran barracks can lift up and move from current location.

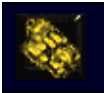

a) Moveable units.

These units play a main rule of RTS game. They accomplish most works in the game, collecting resource, attacking opponent, and scouting over the map. Beside basic movement function, they could attack, patrol, stop, and hold. In addition, it may have extra functions due to different units. For instance, workers have collect resource, repair, and building. Tanks have an anchoring function. All this function presents as a blue print. F1 presents a sample of blue print; it's a terran worker in Starcraft



Fig1 blueprint sample [4]

Look deep in this kind of units, we take unit tank in Starcraft as a n example to view detail features.

Tank	 Normal	 Anchoring.
Mode:		
Hit Box	Large / 31*31	-----
Supply	2	-----
Cost(crystal/gas)	150/100	-----
Energy	0	-----
Building Time	50	-----
Armor	1(+1 upgrade)	-----
Hit Point	150	150

Attack Power(ground/air)	30+3/0	70+5/0
Attack Type	E	ES
Splash Damage	0	Radius=10/25/40
Attack Cool Down	37	75
Attack Range	7	12
Sight	10	10
Move speed	1.8	0

Tab1 feature of Tank in Starcraft. [4]

In this table, the information is unperfected. But still it presents the most important features. In addition, we should notice armor type and attack type. There is always a restrained relation between these two features. Besides the normal attack power, a specified attack may reduce or increase its damage due to the target's armor type. In Starcraft, it works in another way, armor type is equal to hit box. Attack type E has additional damage to large Hit Box, and only 50% damage to small one.

b) Buildings.

Buildings accomplish the crafting and upgrade technology. Some of them simulate the contravallation. Also the technology tree present as a building requirement of buildings. (More details see 2.1.2). Normally, the blue print of buildings only includes the units that can be craft and technology upgrade options. In order to make it clear for players, Starcraft asunder them.

2.1.2 Technology tree

It is an important conception in RTS game. It is an abstract hierarchical visual representation of the possible paths of research that a player can take. [5] In a simple words, at the beginning of a RTS session, player only have a few options to building and craft. Player starts at root of the tree. After player finishes a specified research, it opens more branches. Player gets more options to craft or build. Analysis of a tech tree can lead players to memorize and use specific build orders. Actually, how to and when climb the technology tree in RTS. It is a real challenge for both players and AI.

Fig 2 shows the technology tree for Terran race in Starcraft. It presents the relationship of buildings. You can easily find the requirement of each buildings and its attachment. For instance, the airport requires factory; to building factories, it needs a exist barrack. In addition, moveable units have same kind of technology tree.

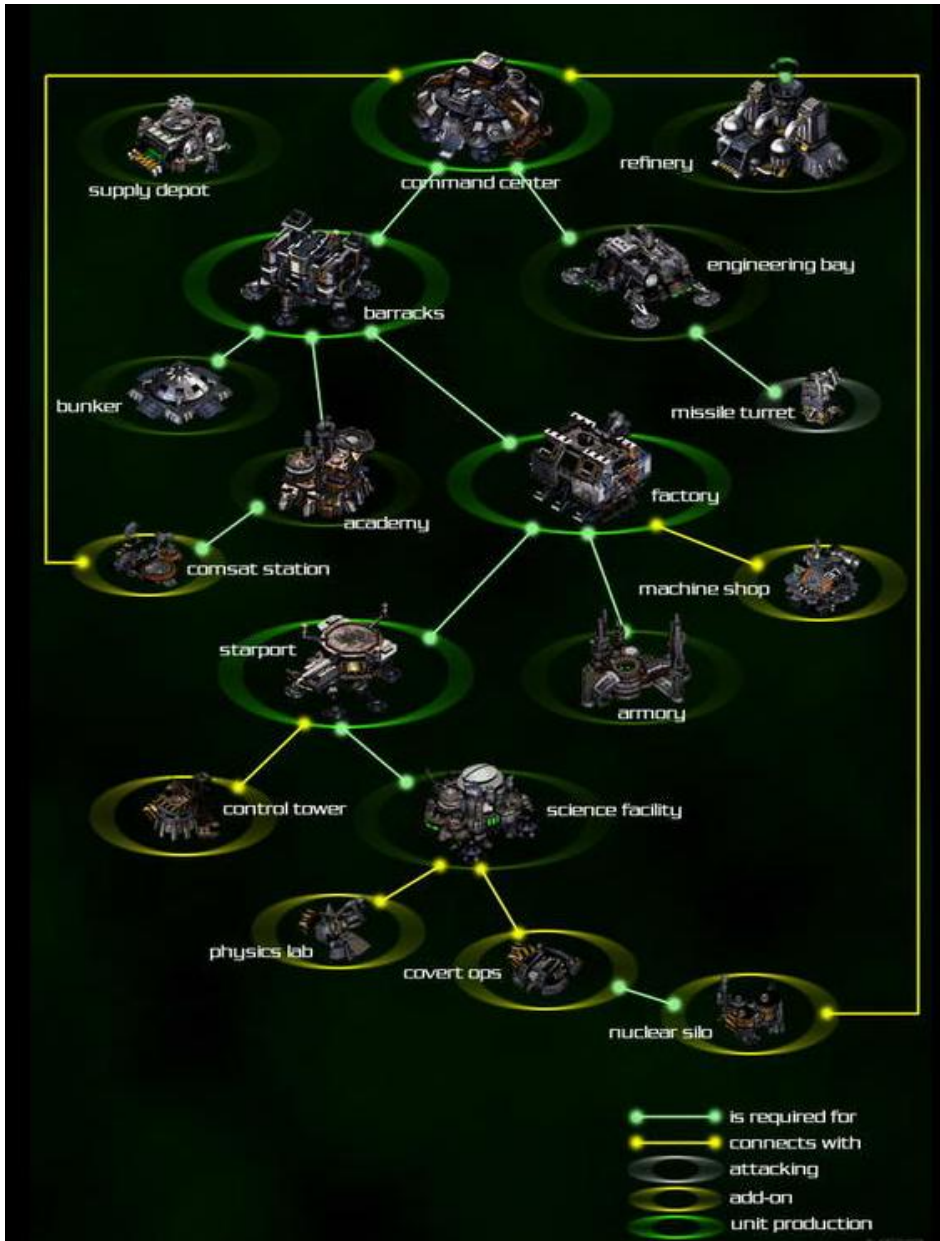


Fig 2 Technology tree of terran race in Starcraft [4]

2.1.3 Terrain definition.

Beside the features we presents above, RTS game has features to define the terrain information, such as trees, highland, lowlands, rivers and pools. They simulate the real world terrain. In the earlier games, this kind of features is used to making the terrain diversiform. They don't affect the battles. Later, game developers make them genuine. They could affect the sight of game units. For example, a unit located at the edge of high land, it may have a unit in lowland. But the units in the lowlands can see it. Even, they can affect the attack power of units. In Starcraft, attacking units on highland from lower area, it has 50% miss chance in attack.

2.2 ORTS

As we mention in chapter 1, commercial RTS game close its resource code. Researching based on those games is inconvenient. We need an open source environments for our research. ORTS is the right one.

ORTS is short for open real time strategy. It is an open environment for RTS designers and funs. “It is a programming environment for studying real-time AI problems such as path finding, dealing with imperfect information, scheduling, and plannin g in the domain of RTS games”.[2]

It provides a standard RTS game, contains everything that you can find in commercial ones. Also, you can use blueprint files to define you own game. The whole system is implemented using C++. ORTS use C/S mode instead of peer to peer mode. Generally, a server simulates game, and clients receive current game view from it. AI design in ORTS is free; there is low level behavior. Client is totally controlled by designed AI. Developers can investigate partial part of the game AI, like path finding; also they could stand on a high level to design the whole AI using some exist library. Our research belongs to the second one. Tab 2 gives an overview of ORTS characteristics.

ORTS characteristics	
License	Free software (GPL)
Topology	Client/Server, server simulates game, sends the game view to Client. Fair game.
Communication Protocol	Open.
Game complexity and specification.	Users can define their own games using blueprint (.bp) file, when game starts, Server reads the game and world description from .bp file.
GUI	It provides both 2D and 3D viewer.
Unit control	Parallel, free define.
AI	AI is local to each client.
Remote AI	Design you AI in client.

Tab 2 ORTS characteristics [2]

More details see ORTS Homepage

<http://www.cs.ualberta.ca/~mburo/orts/#Overview>

2.3 AI in RTS.

AI is the essence of intelligence that controls intelligent agent, where intelligent agent is a system that perceives its environment and take actions to maximize the benefit to the certain goal. [6]

In RTS, AI can divide into two layers. At first, AI controls and optimal basic behavior. For Instance, path finding, collect resource. It is a low level AI affects single unit. The second layer, AI works in macroscopically, it “thinks” the strategy in general. AI implements scheduling, planning and making decision to take the right and suitable actions, Such as when to craft units, when you upgrade technology and so on. Furthermore, some exist AI packages have temporal reasoning functions . F3 presents more detail of this two layer conception.

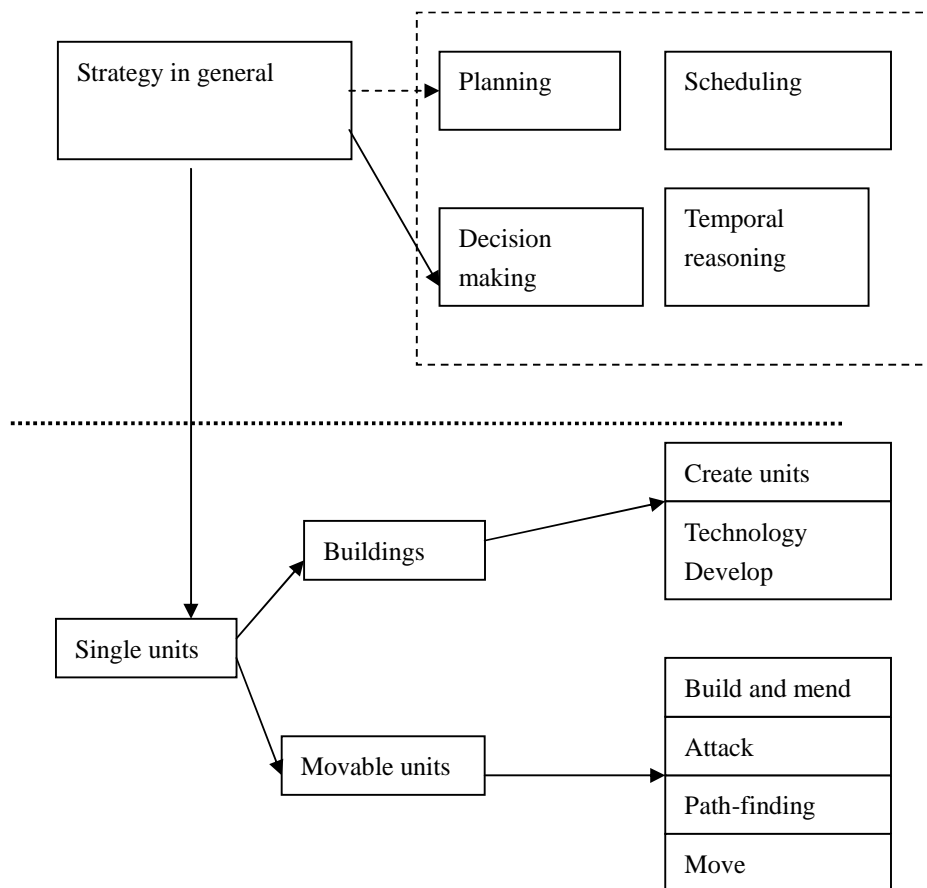


Fig 3 Detail of two layer conception

In the following, we analyze exist commercial RTS game logic and find AI mode hiding in it. As a professional Starcraft (SC for short) player, we will explain the AI conception based on the game SC. First of all, some kinds of resources scatter in the

map. They should be collected for future using. Such as technology developer, buildings, units craft. In SC, we have crystal and gas. AI should have a strategy to discover them. As we develop our economy in real world, player needs balance the number of work and the number of resource point. In addition, there is a balance between develop economy and develop technology. Secondly, there is always a conception that different units may have extra damage to one specified type of units. In this case, we need to **scout** the set up of opponent troops. So our crafting gains more pertinence. It is impossible to find a setup of troop that is invincibility. A bad set up will lead player to lose. Hence, scouting is a key to win. At last, as a simulation of military tactic, RTS game is full of conflicts. These may occur any time during the game. Consequently, AI has a strategy of control its troop. This package named battle control. Even you have more powerful troop and more resource and other predominance, a bad control in a key battle, you will lose the whole game.

Sum up, we abstract an AI mode. It is a mode in high level, the strategy in general level. Fig 4 presents its relationship among those packages.

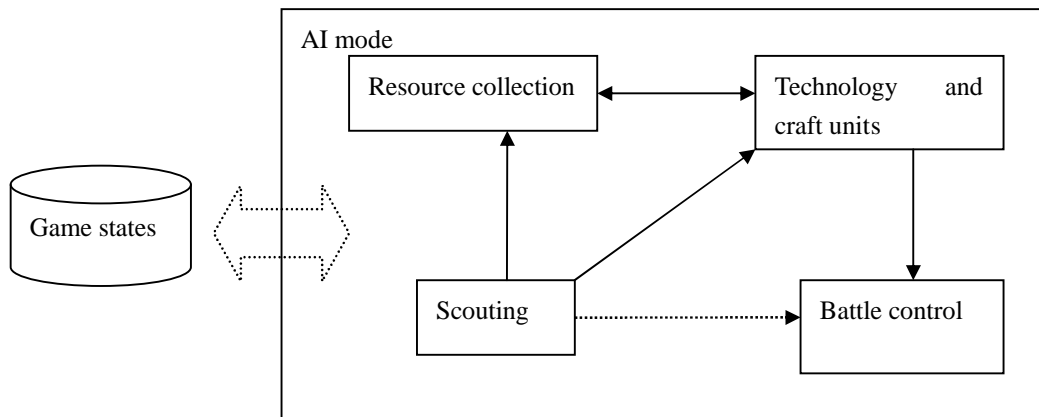


Fig 4 AI mode

There are 4 packages: scouting, resource collection, technology and craft and battle control. In the following part, we would present some basic description of each package.

Scouting package implement the observing function. It defines a strategy, which let AI player explore the environment in suitable time and maximize the information. The main aim is find opponent control center, and scout their developing process and troop set up. Because there is war fog¹ around the map, scouting is processing all the time.

¹, War fog: it is a conception in RTS game. Player explored an unknown place; after the units leave, fog comes. Then we can saw the terrain but only terrain. Unit action is hided, if no friend unit located there. We call this fog for hiding is war fog.

Resource collection defines a serial of action to finish basic resource collection. Beside the basic functions, it contains a strategy to balance the economy and crafting. Including, when to build more control center and more works.

Technology and craft units, this package implement the upgrade process and crafting process. It decides when and what to crafting, and upgrade. Decisions making is based on the game state. It works with scouting package.

Battle control, It defines the commands while in conflicts, including movements, attack sequence. In F4, there is a dashed collect to scouting. Scouting effect battle control indirectly, it only decide when and where to fight.

2.4 Relevant learning theories

Most AI packages in commercial games are lack of planning, and learning. It is the reason that why current human player are much better than AI. Actually, In AI research and other relevant area, learning automaton is highly investigated. Due to the special environment in RTS games, we need to choose a suitable method. In this section, we introduce some methods, that may used and discuss in our research.

Before introduce these methods, we presents the definition of learning in computer world. Learning defined as any relatively permanent change in behavior resulting from past experience, and a learning system is characterized by its ability to improve its behavior with time, in some sense tending towards an ultimate goal. [7] For example, consider a mode that a finite number to actions should be done to reach a certain goal. In the beginning of this system, we don't know the best sequence of these actions. It takes actions in a random manner, for each action, system response in a binary manner indicate whether the selected action is good or bad. System iterates this manner; obtain a final evaluation of a sequence. Under these circumstances system should plan a choice of a sequence of actions and process the information so that it learns the best sequence.

After definition of learning, we start to introduce some learning methods. In addition, some related conceptions are presented. These conceptions may not directly used to solve our problems. But it works like a mirror to let your know which is special in our design. In addition, you could have some basic ideas while we discuss the reason why our design is more suitable than others.

2.4.1 Learning automaton [7]

Learning automaton is an abstract name of those learning methods. It presents the normal definitions in this learning area, including conceptions and normal behaviors.

It contains three main parts. At first part, we introduce stochastic automaton; secondly, we describe the based environment of learning automaton; at last we give a prospect of processing in learning automaton.

a) Stochastic automaton

In stochastic automaton, it defines a sextuple $\{x, \dots, p, A, G\}$.

x	Input set (0-1).
	Internal states
	Output, or chose actions
p	Probability vector governs the choice of state of at each stage
A	A is algorithm of update p (n+1) from p (n). (reinforcement scheme, see 2.4.3)
G	An output function \rightarrow .

Tab 3 sextuple of stochastic automaton.

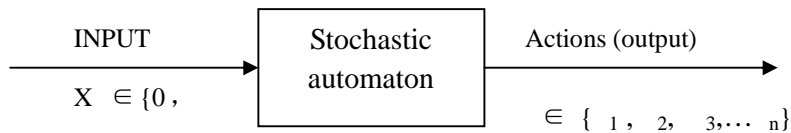


Fig 5 Stochastic automaton

Take a student-teacher pair for example; student answers teacher's questions in a random manner. Due to student answer, teacher responses in a binary manner indicate whether the answer is good or not. X denotes the response of the teacher. indicates the teacher's state. Because of current states and response of teacher, student answers a certain answer. The certain answer is decided by p . now it's clear to see function G in general stochastic automaton define a manners to choose the next action and decision due to the past information.

b) Environment

In this case, environment is unknown and unpredictable. We only interest in the environment with a random response characteristics in this problem consider (shown in Fig 6). An actions influence the environment as an input. Response due to the actions is output. Response in a binary manner, it belongs to a set x from 0 to 1. Zero is a nonpenalty response or a reward, one is penalty response. C_i denotes the probability of penalty depends on the input. Environment is stationary if C_i doesn't depend on n . otherwise, it's nonstationary.

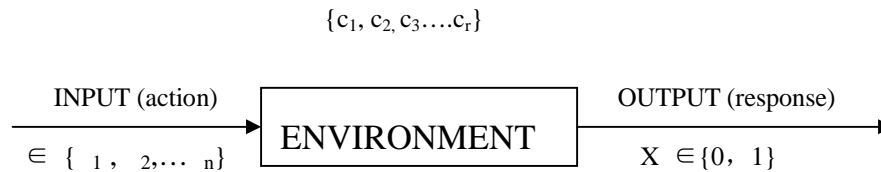


Fig 6 Environment

c) Learning automaton.

Connecting automaton with the certain environment, the action of automaton is an input to environment. The response of the environment sends to automaton. In these turns, responses influence the updating of the probability of actions. In a initial view, response and probability vector $p(n)$ both are random.(connection shows in F7)

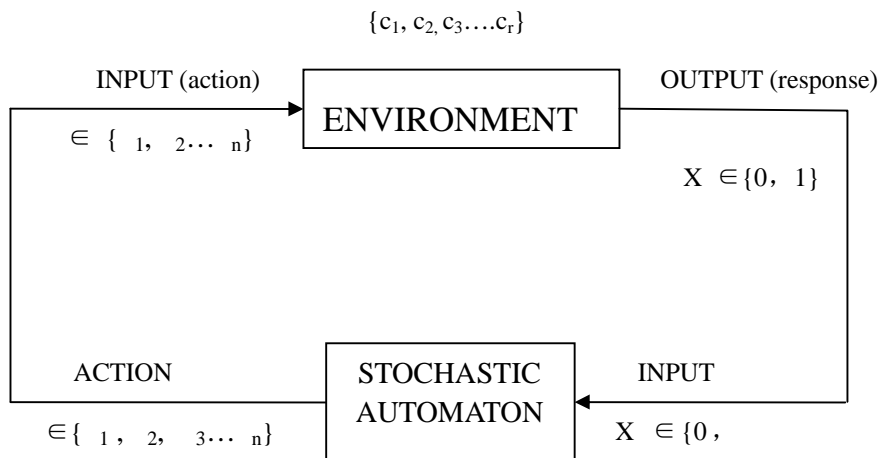


Fig 7 Learning automaton

“Learning automaton is a stochastic automaton that operates in a random environment and updates its action probabilities in accordance with the inputs received from environment so as to improve its performance in specified sense”[7]. A system may have some learning automaton. They work together to reach a certain goal, a final response from environment. System analyses the response and using a recursion method to update $p(n)$ in each automaton.

Mode of learning automaton can be divided based on the response types of environment. The first one is called P-model, environment response in binary manner. The input set only have $\{0, 1\}$. Otherwise, it's a Q-model, which input set is a finite collection of distinct symbols. These modes appear appropriate in specified situations.

2.4.2 Decision tree

In machine learning area, a decision tree is a predictive model. It maps all possible way from observations of system. Different branches represent variety of decisions. It starts from a root, a serial of decisions lead the system goes through branches to reach a final leaf. It uses a kind of choosing manner to fill of decision tree. Analysis this tree, an optimal sequence of decision is learned from it. F5 presents an example of decision tree with some following comments.

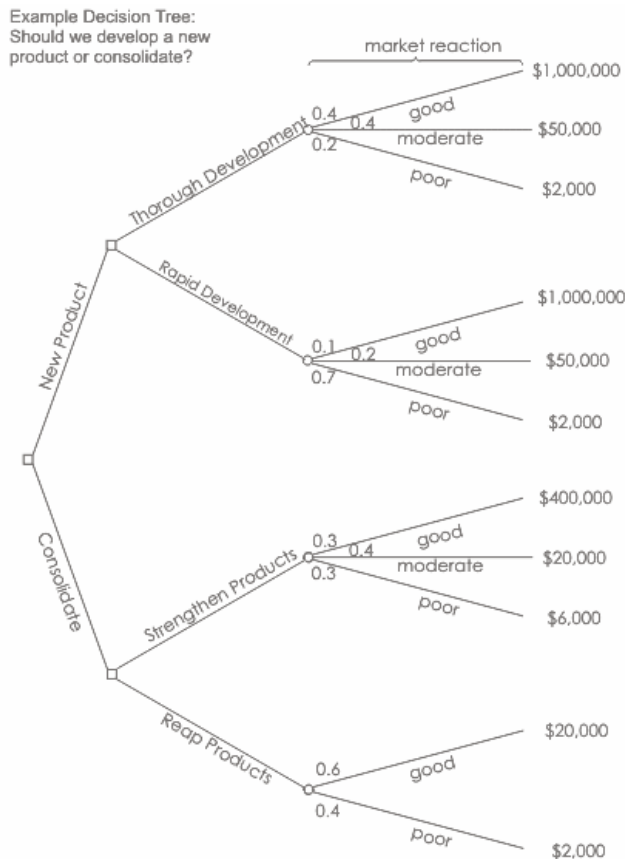


Fig 8 Example of decision tree [8]

It is a decision tree to decide whether develop a new product or improve one. It starts with a root, a decision is branch. Try to draw the possible decisions to full of decision tree. The original tree is show without values. Works in this tree manner would be done and the possible outcomes are gained. In initial states, the value of tree and even the branches are not uncertain, you need iterate works to full of it.

Once, we have worked out the value of these outcomes, and the probabilities of the uncertainly response. It starts to calculate the value to helping decision making. Take

values in F8 as an example. The branch “thorough product” gets a value of \$420,400. F9 shows how to calculate and the tree after calculated.

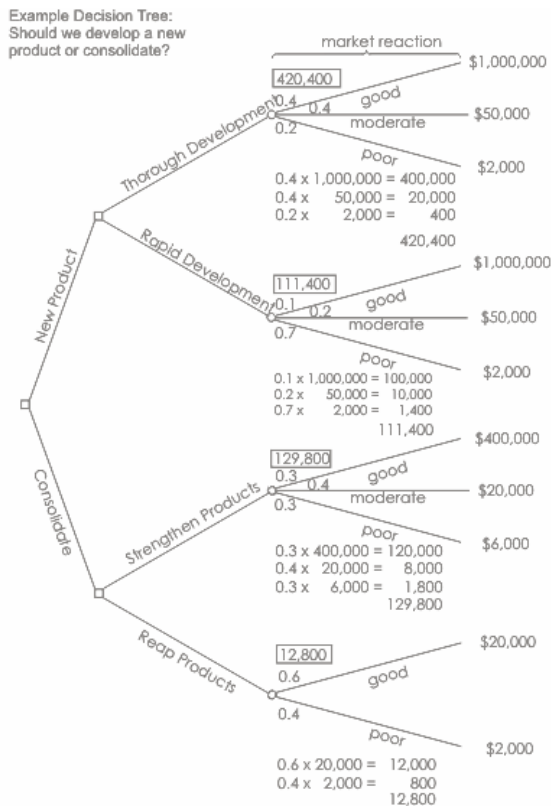


Fig 9 Decision tree after calculated [8]

By applying this technology, we easily research a result that the best decision is “new product”. The processing of this tree can be thinking as a kind of learning. The uncertain information is learned from its environment and decision making finally depends on these information. In some situation the value of the nodes may changes from time to time, so values are updated during this period. Always, it choose s the current best decision.

Decision tree have some advantages. It’s a graphic represents decision alternatives, possible outcomes and chance events schematically. It’s clear and intuitionistic. Second, it is efficient; you can quickly map a complex decision problem in such a tree very clearly. Also it is convenient to modify the tree while new information becomes available. [11]

2.5 Other related conceptions

In this section, we present other conceptions used in this AI research.

a) FSM

Finite State Machine, it is a basic foundation machine learning and AI. In Wiki pedia, FSM defined as a model of behavior composed of a finite number of states, transitions between those states, and actions. [9] For modern computers, we can consider it as an exceedingly complicated machine. In order to understand the essential nature we skip all the complexity and consider the simplest machine mode. It's the original FSM. [10]

A **finite state machine** consists of the following:

- (1) Set I called the **input alphabet**;
- (2) Set S whose elements are called **states**;
- (3) Function $T: S \times I \rightarrow S$ called the **transition function**;
- (4) A particular element, $s_0 \in S$ called the **initial state**;

The functioning of the machine is as follows:

The machine starts in the initial state s_0 . The input is a string of characters from the input Alphabet which are read one at a time (from the left). At each stage the machine is in some state $s \in S$. If the machine is in state s , and the next input character is $c \in I$, the machine moves to state $T(s, c)$ and awaits the next input character. The process continues in this way until all the input characters have been processed. [8]

For instance, we can input set $\{1, 2\}$ and four states A, B, C, D. Take A as the initial states. Output is based on the input and currently state, the set of output is $\{0, 1\}$. A figure presents below.

$I = \{1, 2\}$; $S = \{A, B, C, D\}$; $s_0 = A$.

T is given by the table:

	1	2
→A	B	C
B	C	D
C	D	A
D	A	C

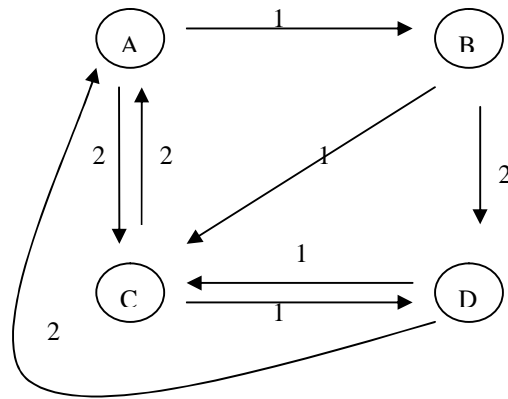


Fig 10 Sample of FSM

Soft agent can be considered as a FSM too. It observes the environment and collects the information from environment. It takes information as an input, due to the input it makes an action (transaction) then change to another states. Then Re-roll this manner.

3. Problem description and analysis

In original project description, the problem is designing a rush-attack AI mode in ORTS which has some learning ability in sense. First of all, it's an AI mode for RTS games. As we mentioned in section 2.3, it should satisfy the specified RTS game due to the game features are quite different from game to game. Secondly, ORTS is the design environment. For the characteristics described in section 2.2, ORTS have much more convenient for RTS AI researchers. Before the "AI mode" there is a modifier: rush-attack, which denotes the main strategy and goal in this AI. At last, this AI should implement the learning ability. This is the most important part. It requires the AI automatically change its decision making process depends on the responses of the game result or other features.

In section 3.1, we describe the sub problems in detail following the presentation above. Hypothesis and feasibility analysis come follow. We would change and drop some undoable part, and redefine our precise research description. At last, it's a short presentation of except outcome.

3.1 Sub problem

Sub problem 1: This is an AI research in RTS game. Due to the features are different, the AI will be different from game to game. *What kind of RTS games* is used in our research? Although, in the develop environment, there is a standard game type. Still we need to find the key feature of this game. A short analysis of the game type should be done before we start research. It gives guidance to our project.

Sub problem 2: In the main description, *ORTS* specify the design environment. It is open source development for RTS AI researcher. Furthermore, ORTS have a different architecture for RTS games. Unlike the normal peer to peer mode, it uses a client/server mode to implement the real fair game. In this case, how does ORTS work in this architecture? For more details, how ORTS implement RTS games? What does it provide to AI researcher?

Sub problem 3: what modifier "*rush attack*" means, and what kind of AI can be considered as a rush-attack AI. In literal aspect, rush-attack means attack your opponent as fast as you can. It is a complex definition. We don't suppose to develop under this definition. In the aspect of RTS player, they plan of how to do a rush-attack? Players may develop economy first, or crafting troops as soon as possible, also they may choose a strategy between. This problem seems easy, players can change their idea based on the game states, then defeat his opponent quickly. Could an AI deal with this changes as human player. It is hard to answer; we must consider a way to redefine the conception in order to make it simple and easy to implement.

Sub problem 4: As a human player, we can easily learning from the past game experiments and improve our playing skills. *As an AI, how and what to learn?* In another words what kinds of learning method is the most suitable for RTS AI. We would like to implements some previous methods. Also, based on a chosen method, what kinds of feature are useful? The decision making processes, generally strategy or some other things that can influence the game. In addition, for single researcher, I would like to simplify the problem so that I can handle.

3.2 Simple solution to sub problems

We give simple solution and short investigate to each sub problem and feasibility analysis follows each hypothesis in this section.

3.2.1 Type of RTS game [12]

The type of RTS game may influence the AI mode. So an AI solution should base on a specified game. As our research environment, ORTS allows research to define its own game features of RTS game. It uses BP files as the definition of such games. But we are not game designer. Fortunately, it also provides a standard game type. It locates in file orts\trunk\testgame. All the information is defined in BP files. Let go deep in this information, and find out what kind of game it is. We should notice that the game logic of RTS game almost the same, the different is the game units, terrain type and other features may influence the game result.

The standard game defines in 9 BP files. (F11 shows the file system) this game relates to Starcraft. Most of his feature inherit form terran race in Starcraft. Analysis of these BP files helps to understand the game and find its characteristics.

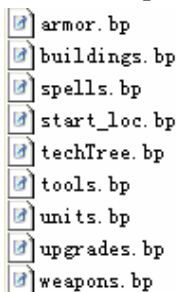


Fig 11 BP files of standard game

a) Armor and weapons:

Like we described in section 2.1, this standard game have armor type and weapons. There are some restrains between armor and weapon. These kinds of information are defined in armor.bp and weapons.bp. F12 (a) presents a sample definition of armor, (b)

shows a sample definition of weapons.

```
#-----
class terran_medium_armor
  has base_armor
  set AC 1
  setf slashing 0
  set piercing 0
  setf explosive 0
  setf blunt 0
end
#-----
```

(a)

```
#-----
class tank_turret
# is targ_weapon
# is area_weapon
is splash_weapon
set min_damage 28
set max_damage 32
set bonus 1
setf damage_type EXPLOSIVE
set max_air_range -1
set max_ground_range 7 * 16
#set min_ground_range 2 * 16
set cooldown 37
set splash 5
set upgrade_bonus 3
end
#-----
```

(b)

Fig 12 Sample definition of armor and weapon

Most information is similar with Starcraft. But we don't find the words to definition the restrains of each armor type and weapon type. It uses another way. In a n attack function, the code shows this restrains. (F13 codes form weapons.bp) The value of specified armor value could reduce the corresponding weapons type.

```
if (damage_type != NORMAL) {
  if (damage_type == SLASHING) shield_dmg -= targ.shielding.slashing;
  if (damage_type == PIERCING) shield_dmg -= targ.shielding.piercing;
  if (damage_type == EXPLOSIVE) shield_dmg -= targ.shielding.explosive;
  if (damage_type == BLUNT) shield_dmg -= targ.shielding.blunt;
}
```

Fig 13 Codes form weapons.bp

b) Units and Buildings

These two BP files define the feature of units and its actions. The feature definition is following the style in F12. Unit's actions are defined as function in unit class. There is an example of it.

```

action anchor(;;) {
    if (this.mode != this.NORM) return 0;
    if (!this.can_siege) return 0;

    this.weapon.use_big();
    this.is_mobile -= 1;

    this.mode = this.ANCHORING;
    this.anchor_step() in 1;
}

```

Fig 14 Example of action definition (tank anchor)

c) Sum up:

After viewing all information, the standard game inherits most features from the terran race in Starcraft. There is a few differences, we present the difference may influence our design. At first, Armor and armor restrains works in different way. Secondly, the technology tree is a bit different (discuss later).

3.2.2 ORTS develop environment

In this section, we would give a high overview of ORTS; then pick out the useful information for our research. Our working environment is windows.

a) System overview [1]

We start with the system overview. ORTS use a client/server mode. When starting the server, it reads the game information in the BP files together with terran description and initial unit locations (F11, start_loc.bp). After this processing, the server holds on, and waits for all clients connect to the server. As soon as all connection established, server sends game description to them, and enters the simulation loop. In these loops, clients only got its individual world views, actions based on these unperfected information sent back to server. Due to the actions of each loop, the views are updated and the game state accordingly. Client and server repeat this loop till the game ends. (F 15 shows the processing)

ORTS already provide critical components to implement the object vision, motion and collision computation, as well as data transmission protocol. These kind s of efficient algorithms are presented in [13]. Furthermore, the source library provides most basic

functions, like path finding, moving, and so on. These are contributed by previous researchers.

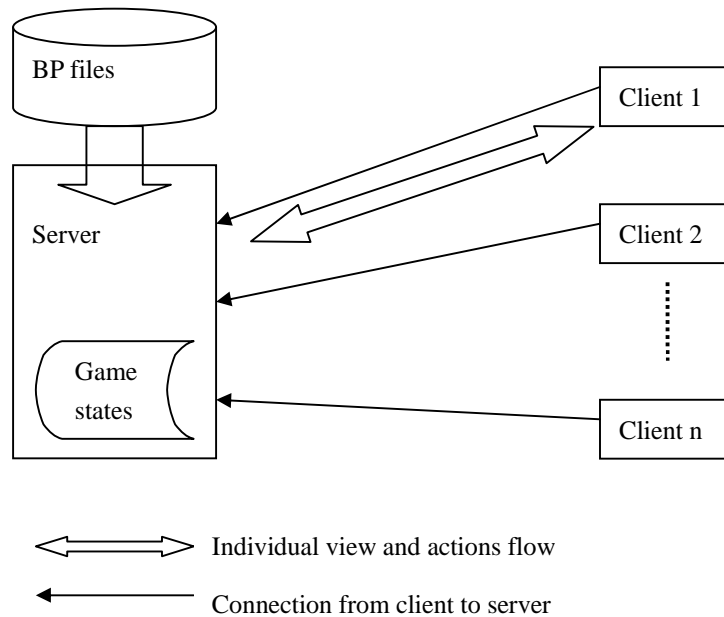


Fig 15. System overview of ORTS

In addition, it provides a 3D graphic interface for games. Distinguish to commercial games, it give a widely world view. In my opinion, it is a convenience to AI designer; we don't suppose to change view frequently to check my AI is works well or not.

b) Develop detail

After the overview, we look deeper in this system and introduce develop detail of ORTS for developers.

First of all, ORTS simulate RTS game in 8 frames per second. A loop function will loaded both in server and client. In this case, client could communicate with server and synchronization. Once per frame, server sends the individual view and client can have maximal one actions per game object that under their control. Then client send the action back to server. These actions then randomly shuffled and executed in server.

Secondly, AI is located in each client. For developer, getting start with your own AI is very simple. If you use windows (we use windows in our research). ORT provides a VSTOOL located in `\orts\trunk\misc\windows`, compile it, and then we could use it to create an empty client project in VC++. Secondly open this project, find the method `computer_actions()` located in `SampleEventHandler.C`. This method is automatically

loaded in each frame (loop), and it's the starting point of the AI development.

Then we can have real start in ORTS. There is a sample code show in F16.

```
//This is the game object we use to read the
//current game state
const Game &game = state.gsm->get_game();

//This is the unique id for this player
const sint4 cid = game.get_client_player();

//Get the objects for the player with id 'cid'
const Game::ObjCont &objs = game.get_objs(cid);
FORALL (objs, it) {
    GameObj *gob = (*it)->get_GameObj();
    //Check if the object is in the game
    if (gob && gob->sod.in_game && *gob->sod.x >= 0 && !gob->is_dead()) {
        //Get type of object
        string objType = gob->bp_name();
        //Get the position of the object
        int x = *gob->sod.x;
        int y = *gob->sod.y;
        //Get the unique id for this object
        PlayerInfo& pi = game.get_cplayer_info();
        uint4 objId = pi.get_id(gob);
        if (objType == "worker") {
            int maxSpeed = *gob->sod.max_speed;
            move(gob, x + 1, y, maxSpeed);
        }
    }
}
```

Fig 16 sample code. [14]

In this sample, client get currently game states then pick out the friend object by using the client id. For each that in game object, client gets its type and position. For workers, client set action to move one step to the right. [15]

Till now, we get a simple conception of how to develop in ORTS. It is convenient and powerful.

3.2.3 Rush-attack.

Rush-attack is a popular strategy in RTS game. Before introduce this conception, we'd like to discuss a common RTS player's game strategy. As a semi-pro RTS player, most players in solo game, they would develop their economy and technology first and only craft a small number of low level units for basic defense. There a re main troop set up will be middle level units and high level units. This troop setup is powerful but it takes time. It has an obvious weakness, in the beginning of this strategy, majority of resource is spending on development, and its defense is very weak. Rush attack is a corresponding strategy to attack this weakness.

Rush attack means crafting a troop with mass low level units, and attack your opponent's base as soon as possible. The aim is destroy the key buildings before they are able to craft powerful units for defense. Although, it may leaves the rusher's force greatly weakened, and a bad economy situation. Actually, in a perfect executed, this strategy has more advantages then disadvantages, even win in the first attack. The gambit is to slow or cripple the opponent's ability to expand and upgrade his forces. Destroy of key buildings takes a lot of time to recover. In this sense, rushers get time to develop its economy and keep ahead in technological levels and production capability. Finally rushers reach the victory.

In the following part, we analysis the strategy and try to find the basic conditions of rush attack.

a) Information

In RTS game, information is very important. Actually, player's strategy is based on how many information he got. As we presents in F4, scouting which gives more information to player, leads other packages to work. In Rush -attack, information is a key. Player drops its economy and technology development, and gamble in rush attack. In this sense, when and where to attack, also the number of units in the troop is the most important. A bad attack may lead him to lose. How to make these decisions? He needs information to decide the appropriateness of rush. In a word, more information, more wins.

b) Distance

This is an impersonality element. It affects the result very simply. In order to make the rush more effective, player tries to limit the opponent ability to recovery. It means the longer rusher spends in transit, the longer defender can prepare countermeasure.

c) Balance in development and crafting.

Rushers drop development to craft troop. But still, they need basic economy and technology. Then there is a balance between these choices. Furthermore, because of the tech-tree, the troop requires some prior research. Rushers must be proficient in building order so that rush may come even earlier.

d) Macro-control in conflict.

Beside those prepares, the final goal is defeat you opponent in conflicts. This condition is not only for rush attack but also for all RTS strategy. Conflict is unpredictable, there isn't a true power. [16]

3.2.4 Learning method

The choice of the learning methods is based on what goal we want to achieve. Those existent methods already presented in section 2.4. Our design will present after the requirement specification. See section 4.2.

3.3 Requirement specification

In this section, we define the experiment goal of this project. For each point in the goal, we give a short analysis and following a requirement specification.

3.3.1 Experiment goal

Our research tries to design an AI package with learning ability. It implements a rush attack strategy and learn the best troop set up during its game process and the result of each game. In this case, we focus on the building order and crafting decision during the game, and find a most efficient method to “learn”. In addition, a relationship between game states and decision need to be investigated.

3.3.2 Game Specifications

The whole RTS game is so unpredictable; many features influence the game result. It needs to be simplified to satisfy a certain goal. A specification give following, it would help in testing some feature.

First of all, we need discuss the beginning stage of the game. Normally, players would have a certain number a worker (basic units) and a control center. We follow this style. Our AI will choose the game style one in ORTS; it starts with 4 works and 1 control center.

Secondly, in these kinds of AI research, resource collection is a complex problem. We need find a balance in required gathering speed and numbers of workers. Furthermore, what time to expand economy and found a new control center need to be considered while the game going, this decision is hard to make. It needs a well temporal reasoning ability, and excellent corporation of each AI package. It is not the important part of our design. It doesn't worth of developing such a complex problem. In most commercial games, the AI just doubles its resource per gathering. Following this style, we declare that no resource collection is need, the resource increase stably number, which is predefined.

In order to simplify the game and focus on the crafting decision, scouting is defined in

another way, no war fog in this game. In other word, the opponent is exposure to player and no scouting is needed.

We only use three kinds of attacking units in our game. They are marine, motor bike and tank. In Starcraft, these kinds of units have obvious restrains to each other. After analysis the game type in our research. A table shows the comparison of these units.

	Marine	Bike	Tank
Level	Low	Mid	High
HP	40	80	150
AP	4-6	18-22	35
CD	Fast 2*8	Mid 4*8	Slow 5*8
Building time	24	30	50
Money	50	70	250

Tab 4 comparison of marine, bike and tank

Marine is the low level units, with crappy attack power and low HP, but he has advantages in crafting time and cost. In contrast, tank has more attack power and HP, as an opposite feature, the crafting time and cost is really annoyed. Bike is a mid course choice, fine HP, fine attack power. We suppose to find function to calculate its real effective in the conflict.

Specification:

1. Game starts with 1 control center and 4 workers,
2. Resource increase in a certain number in each second . No need to gathering.
3. No war fog in Game, the opponent stats are exposure.
4. Only three kinds of unit will be used. Marines, motor-bikes, and tank.
5. Opponent troop is a static group which already given at the beginning of a game .
6. The game style is the Standard game in ORTS.
7. The final conflict decides the winner or loser. No need to destroy all buildings.

3.3.3 Strategy specification

In general, we use rush attack strategy. AI needs to climb the technology tree, and crafts a certain number of units to defeat his opponent as soon as possible. Upgrade should be considered if needed.

3.3.4 Learning specification

Learning methods would take place in finding the best group set up. In addition, the group is may change in different game; the AI would have some responses due to the changes of opponent group. Furthermore, three kinds of units have relationships that restrict each other. We should dig out this relationship. Decision making depends on this relationship would be more efficient.

4. Design

4.1 AI blueprint

In section 2.3, we describe an AI mode with two layers. Our design abstracts this mode and draws three layers architecture for AI mode. We divide strategy in general level into two layers. AI decisions are presents in higher layers. Before describe our design, we would like give an overview of this three-layer architecture.

4.1.1 Three-layer architecture

General description:

1. AI works on high level commands.
2. Linear analysis, while the game is running, there is a coordinate line of time to show its process. AI is able to make a linear analysis. This mechanism to have a temporal reasoning.
3. Decision make base on a measureable environment. In other words, the system has a value on each object. The higher the point, the more possibility to win. (more detail see section 4.2)

Principles:

We mapping actions and commands into three layers . Single actions defined in **Basic action**. A serial of actions to finish a certain job or function is defined as the **second layer**. In the second layer we sort the serial of actions into 4 packages, **Battle control**, **Scout**, **Resource Collect**, **Tech and Craft**. In highest layer, a learn AI package are located to analysis and make high level decisions.

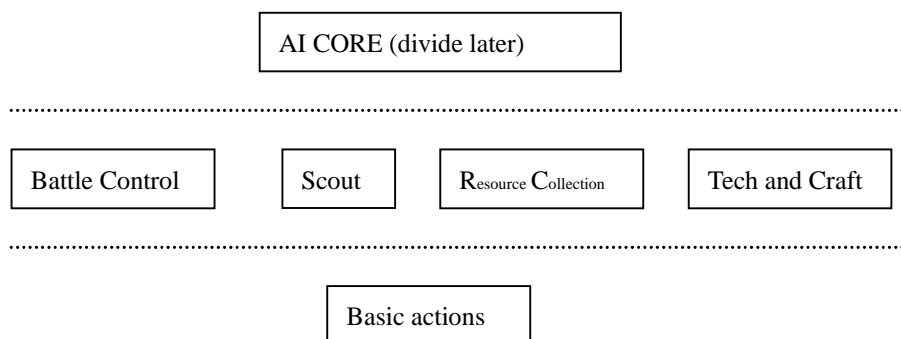


Fig17 Three layer architecture of AI

Description of each package:

1. Basic actions: in this layer, it contains the basic single action, such as move (), attack ()... these actions provide the basic capability of units. They are already defined in ORTS. Furthermore, these actions may fail due to the game situation. Hence, ORTS provide simple error check method in this layer, like collision_check ().
2. Resource Collection: this package defined a serial of commands to finish resource collection. It contains commands to let the works find and gather the resources like crystal and gas in game. It may work with scouting to find the crystal and give an optimal number of works. Also, it is influenced by the number of current resource and game state.
3. Scout: this is a scout method to find opponent base as soon as possible. Once it finds their base, a consecutive scout would be executed to collect more information of opponents. This information includes number of buildings, number of units, possible technology tree, the attacking purpose.
4. Tech and Craft: it defines the commands to upgrade technology tree and crafting if it is needed. In RTS game, technology tree is an important element, the higher you climb, the powerful units you can craft. But in order to climb the tree, a lot of resource is cost; you may fail in defense opponents attack. The AI core would find the balance between crafting and climb technology tree. This package just implements the decision and gives feedback.
5. Battle control: the final purpose of RTS game is defeating your opponent. So you need to attack it, fight with it. Hence, this package is the most important part. It defines the commands while in conflicts, including movements, attack sequence. Battle snapshot sends to AI core as a feedback. After analysis, new control conception would return to this package.
6. AI core: this package works with the second layers, it observe the second layer and make decisions in a high level. We must insist on a conception. AI works on a "thinking" level, it doesn't change game directly. Instead, some fetal variables will sends to second layers. So the packages in second layers can make changes due to these changes of variable.

4.1.2 Our design

First of all, it gives a system view of our design. As we mentioned in pervious parts, our design don't have Scout and Resource collection. The figure below presents the relationship of these AI package. Only short conceptions of each package explained in this section, more detail see specification design.

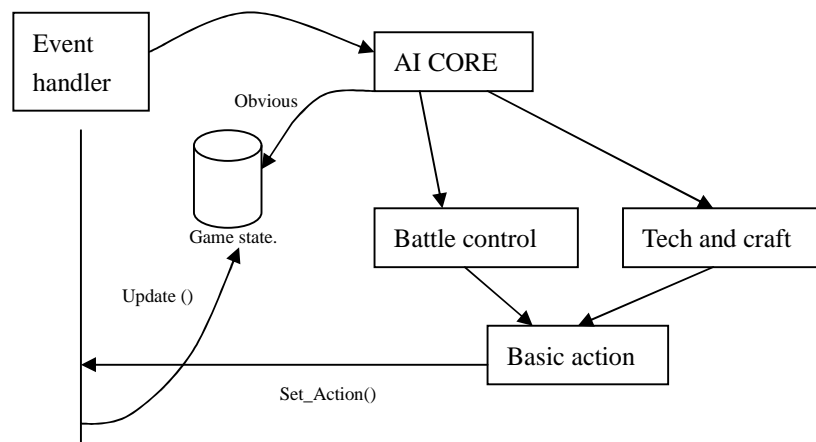


Fig 18 AI system overview

In ORTS environment, `compute_action()` is loaded each frame. It is the entrance point of the design. After the AI obvious the current game state, it makes a general decision. Then the decision forwards to mid layer packages. In mid layers, the general decision is disassembled, and implement by using basic Action package in lowest layer. The serial of basic actions calls the event handler, to set its actions. Some actions may last a few frames. The middle layer package have a strategy to make sure is finished in the right order. Then event handler receive this actions and run them in a randomly sequence. At last, updates the current game state.

Game state:

We store the game state in a date structure. In order to store the all useful information, it contains 4 single variables to denote current money, gas, supply and frame. In addition, 4 four structures denote building situation, workers, my troop and opponent troop.

Details:

1. Single variable.

```

Int c_crystal;    // current number of crystals.
Int cycle;       // number of frame now, time of the game.
Int I_crystal;   //the increasing number of crystal.

```

2. Structure: Buildings

```

{  int ID ;    //primal key, it's a unique ID to distinguish it.
   String name; // the name of the building. It is defined in blueprint.
   Int x, y;    // location of the building.
   Boolean Attachmen;t // if the building have a attachment.
   String C_object; // current object in crafting.
   Int time_left; // how much time till the object finish. Initial= 0;

```

```

    }
3. Structure: Worker.
   {  Int ID;           // primal key, unique
     String building;  // the name of building that worker works on. If the
                       // worker is free, building = none;
     Int timer;        // how much time till the building finish. If builing ==
                       // none;
                       // timer = 0;
   }
4. Structure :a_object;
   {  Int ID;           //primal key,
     Int timer;        //how long time it finish
     Int AP;           // attack power
     Int CD;           // cool down to attacking
     Int HP;           //hit points
   }

5. structure: Mytroop
   {  Int type;         // type of attack units. Type = 1, units is tank. Type =2, unit
                       // is bike. Type=3, unit is marine.
     Int number        // number of units.
     A_object[n];
   }

```

Opponent_troop is the same as Mytroop. They record the troop set up of each side.

In ORTS, It gives a GSM to record the game states, but when you want get information from it. It is inconvenient. That is why we redefine data structures. Decision making will based on this data.

AI core:

This is a complex package. We would like to describe it in a general way. It presents in two parts. These parts are divided by its functionality. The first one is building part, second one is crafting. It only decides to make decision that what kind of buildings and crafting is needed in a certain time. Once, the decision is made, it forwards to second layer.

One thing should be notice that, in our design, AI core package don't contain the battle control strategy and the precise macro actions for each kinds of unit. Instead, core package make the decision that when to attack and other general decisions. In addition, control strategy is defined in battle control package

Tech tree:

Due to our game specification, only three units are used in our testing game. There are marines, bikes and tanks. In this case, the partial technology tree is present below. We use a FSM to accomplish the climbing process, and also the additional buildings if needed

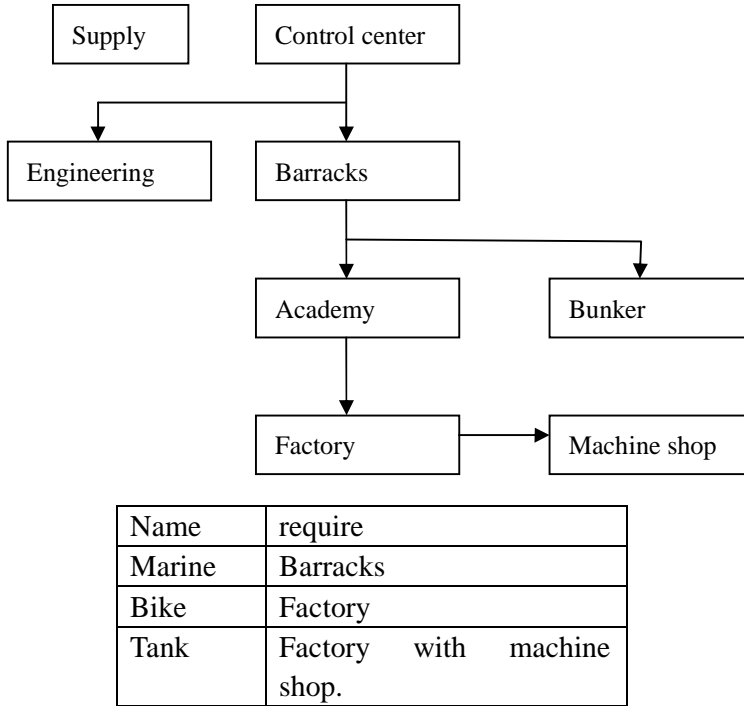


Fig19 Technology tree in our game

In order to craft most powerful units in the shortest time, we should climb technology tree as soon as possible. It has the highest priority. Then building process stops in an unstable state. So the AI should obvious the game state and the money situation to decide that if we need more barrack, factory or nothing. A FSM show its process,

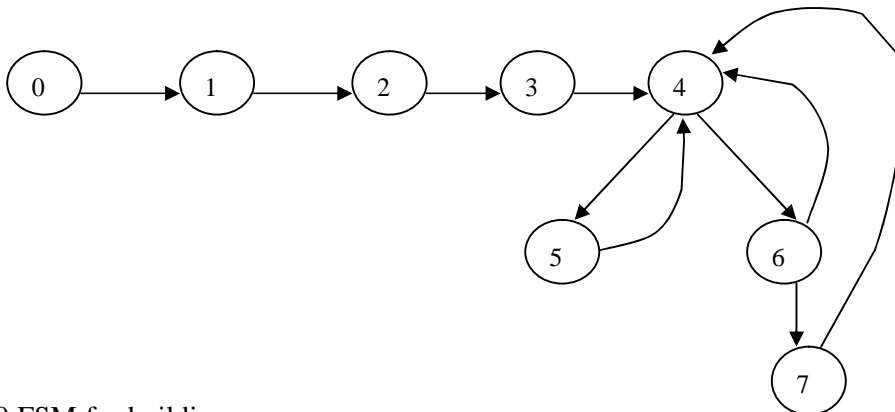


Fig20 FSM for building process.

State 0: the beginning of the game. All resources is saved, because climb the technology tree has the highest priority.

State 1: barracks completed.

State 2: academy completed

State 3: factory completed.

State 4: machine shop completed, technology tree reaches the end. From now on, FSM stops to get ready to receive decisions from AI core. If it needs more barracks go states 5, after the barrack finish, it turns to state 4 again. if it decide to have more factories, FSM go to 6, or 7 it depends on the requirement of machine shop , after finish, turn back to 4. Furthermore, we can craft more than one buildings at the same time.

Crafting

As it defined in project goal, we need to craft the most powerful troop in the shortest time. There are some questions need to be solve. At first, in this AI, we only consider 3 kinds of unit. Which one of them is the most powerful unit? Secondly, the most powerful unit got a universal power? At last, how to decide the crafting sequence? Read the follow sections, you will find answers.

In normal conceptions, the powerful units have high hit point and more damage per seconds (DPS). We quote the definition of each unit in Blueprint; you can easily find the answer. (See Table 5)

	TANK	MARINE	HOVERBIKE
Armor/AC	Medium_armor/1	Light_armor/0	Light_armor/0
Weapon/damage	Weapon_manager/28-32	Assult_rifle/5-7	Grenade_launcher/18-22
Damage type	Explosive	Piercing	explosive
Cool Down	37	15	30
HPmax	150	40	80
Supply	2	1	2
Cost(crystal/gas)	150/100 (250)	50/0	70/0
Time	50*8	24*8	30*8
Range	7*16	4*16	5*16

Tab 5 Definition of tank bike and marine.

Obviously, tank is the most powerful unit in this three. But which means a universal power? In RTS game, there are other feature may influence the units too. As we presented in section 3.2.1, the restrains of the armor and attacking type. It has a

function presents its relationship (see Fig13). Unfortunately, in the standard game blue print, marine, bike and tank armor only contains defense value in AC. It means that this restrain has a quite small influence in game. What's more? The differences in attack-power, hit-points and cool-down cause the lost of damage, for instance, a tank attacks a marine, marine dead in 2 shots. But within two shots tank cause 60 damage, its more than marine's HP. The extra 20 damage is useless, it is lost. So the force is not universal, we need a variable to modify its power. This variable should be learned in games.

Now we face the last question. It's very complex for AI to decide which one would like to craft first, second, and so on. The FSM is for every building which can craft units. Factories have higher priority to decide what to craft. After all factories make its decision, the barracks can be taken in process.

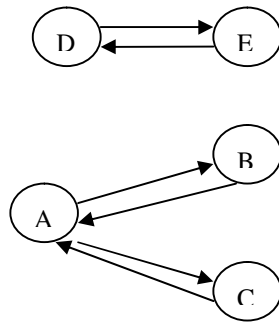


Fig21 FSM for crafting process

A, B, C denote the states in factories, A is the start point. After it decides the units to craft, it changes to states B or C. when it finishes craft, state return to the start point. If it stay in start point, a decision making process is loaded in every game cycle. D, E denotes states in barracks.

4.2 Learning theory

This section has two parts. In the first one we focus on introduce our method of learning. These would explain the reason to choose, and also what features is learned and changes during its process. In the second part, we present influence to the AI system which caused by "learning", and design details of each segments. In another words, due to learning process, some key features are change to fix the situation. Those segments make its decision depends on the current features.

4.2.1 Main learning method.

Before introduce our design, we'd like to explain an evaluation system that widely used in commercial RTS games. Everyone who has played RTS game like Starcraft may have experiments. After the game ends, the view automatically changes to a statistics pages, which presents the evaluation of player performance in the last game.



Fig22 Sample of evaluation in war craft 3

Normally, the winner got higher points. In Fig21, player bibivsvivi wins, its total score is much more then the loser. As we found in this figure, units, heroes^② and resource are all calculated as this points. Look deeper in this figure, we open the “units” column (Fig23).



Fig23 “Units” column

We find some new features, units produced, units killed, building produced, building razed and largest army, all these kinds of information are elements for the score calculation. After a serial of compute actions, finally the system got the scores presents in “unit score” column. As always, winner got higher point.

1, hero is a units conception in Warcraft 3, hero has more force, it is a flag of you troop

It prove a truth that commercial games use a point system for evaluate the game after game ends. It shows a way for design RTS AI. Can we combine the system into the game AI? In this case, the AI could make decision more precise based on this system. A learning method can be located to update the point system. After these kinds of “learning”, the point system finds more accurate value for each element. In sense, AI is “learning”.

Because the safeguard of commercial game, we can't get the resource codes of these system. Fortunately, we find the basic data of Starcraft . It recovers a partial view of this system. At the ends of each unit, there is a score of it. Tab 6 shows the score of those three units which used in our game. This is a evidence of point system.

Type of unit	Score
Marine	100
Motor bike	150
Tank	700

Tab 6 Scores table

Now, we get the main conception of our learning method. It is a strategy to update the point system in our game. Turn back to the specifications, we are using a rush attack AI, which requires product force as soon as possible. In principle, it need s crafting the biggest force per cycle. We abstract an expression for calculating the force of a unit in Rule 1, attack power and the force are direct proportion. Rule 2, the longer units stay in the conflict the more power it have. In other words, HP and force are direct proportion. Rule 3, it's easy to understand that unit has longer Cool Down would reduce his force in some cases. Hence, Cool Down and force are inverse proportion. Follow these rules, we got an expression shows in Fig24. N value denotes rate of lost damage which set as a adjust value for its true force. (See section 4.1.2 crafting part)

$$\frac{\text{Attack power} * \text{HP}}{\text{CD}} * n = \text{Force } i$$

Fig24 Expression of force

The game features in standard game are almost the same with Starcraft. We calculate the force of those three units. A comparison with the point system in Starcraft presented as a simple proof to show if our expression works . In general, the rate should be nearly the same. T7 presents the comparison.

	Score (Starcraft)	Score (standard game)
Marine	100	16n
Bike	150	53n
Tank	700	121n

Tab 7 Scores comparison (Starcraft and standard game).

In Tab 7, the rate between marine and tank is fine. But it seems something effect the force of bike in standard game. Look deeper in it, the armor type in standard game doesn't reduce the attack power as it shows in Starcraft. Bike's attack type does half damage to units with large hit box. In standard game it doesn't work like this. Bike damage only reduced if the target's armor has a value in that type (see section Fig12 (a)). In this sense, bike's force are doubled or even tripled. It explains the difference rate in standard game. Sum up, our expression is works, plus some other features which based on the requirement. Decision making will based on these final score. At the end of this section we introduce the learning process. The decision making processing is explained in section 4.2.2.

The key of learning is finding the lost damage of each unit during the conflict. The precise percentage is updated as the adjust value N. The update process defined in such a way. In the specification, we only have a final conflict to decide winner or loser. It means the final conflict is the only chance to find and update the adjust value N. We need calculate the lost damage of units during the conflict. Different kinds of unit have different N value; also for specified target the N is deferent too. Fig25 presents the expression of value N.

$$N_{(type)} = \sum_{i=1} \frac{Shpi}{Sapi} \cdot \frac{Tn}{Troopn}$$

Shpi: the sum of lost HP, target type i.

Sapi: the sum of attacking power. Target type i.

Tn: number of units that is target type.

Troopn: number of whole troop.

Fig25 Expression of adjust value N.

N is a sum up of percentage of valid damage to different target type multiplied the percentage of the unit type in opponent troop.

4.2.2 Decision making process.

Scenario 1: Decide what to build.

In this part, we present all situation that meet during the game processing. At the beginning of the game, climbing the technology tree has the highest priority. All money is saved for these kinds of using. Second, after the tree is finished, we need to decide whether more factories or barracks.

a) Climb the technology tree.

This is a simple process. As soon as the game starts, process is loaded. It commands the worker to build buildings in a specified order which follows the FSM in Fig20. In each game cycle, this process is loaded to check the current FSM states and make the next decision. Fig26 presents the process.

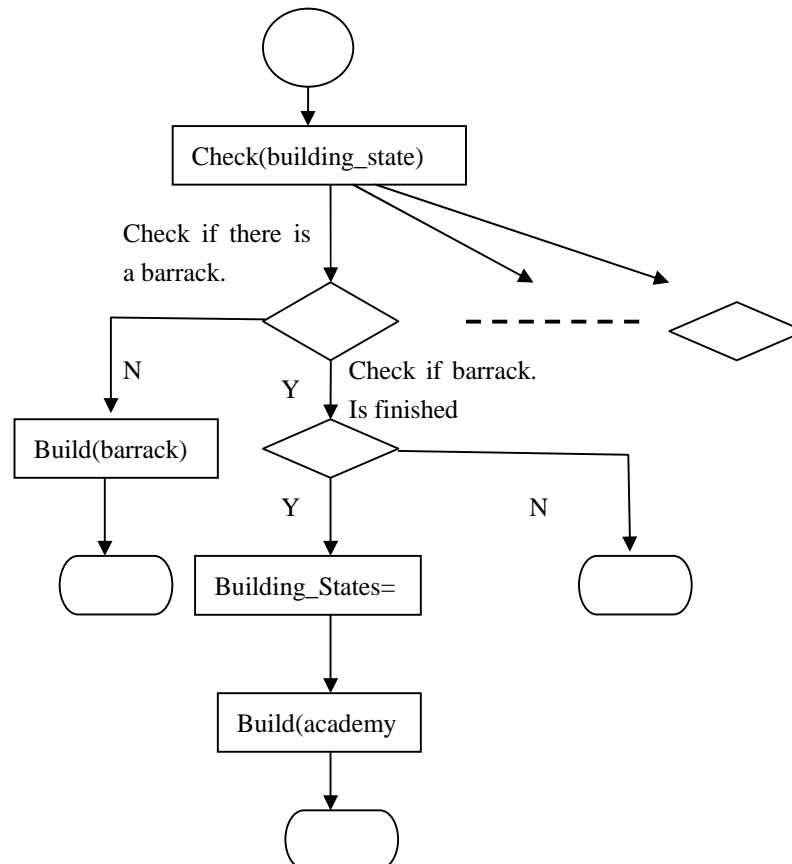


Fig 26 Partial process graphic of building check

Fig 26, only presents the process while building_state=0. The others are the same with it, just the required building is different. For example, if building_state=2, then we

would check if there is factory. If no we set action to start build. Otherwise, if factory is in building, we stay waiting. Else, we change the `building_state= 3`, and build the next building “machine shop”. Tab 8 shows the pseudo code of this process.

```

Building_check() {
  Switch (building_states)
  Case 0:  if (barracks is in building)
            Break;
            If (barracks is completed)
            {  Building_states=1;
              Build(academy);
              Break;  }
            If (no barracks)
            {  Build (barracks);
              break;  }
  Case 1:

```

Tab 8 Pseudo code of building check

b) More buildings.

As we presented in Fig20, when FSM reach state 4, it stops in this unstable state. A temporal reasoning method is imported to decide if we need more factories. Only factories are considered here. Because, compare with tank and bike, marine is useless while the game pass the beginning phase. In this case, FSM is simplified.

In general, we expand our productivity while income in bigger then expenditure. Beside, the general conception, we have a lot of deposit resource while we climb the technology tree. Hence, we use a method to reasoning the expenditure in future. Based on this assumption, AI decides how many buildings are needed. Take tank as a checking unit, we define expression like this. (Fig 27).

$$F_num = 1 + \frac{\frac{Cur_money}{endcycle - cur_cycle} + E_money}{Tank_consume_percycle}$$

Cur_money: current resources.

Endcycle : the final cycle when the conflict starts. In another words, it is the end of crafting process. This have a initial value, and it changes after games.

Cur_cycle: current cycle.

E_money: resource gain in each cycle.

Tank_consume_percycle: the cost of crafting a tank percycle.

Fig 27 Expression of F_num

After FSM reach state 4, AI calculates the F_num and compare with the number of current number factories. If it is bigger than current num, FSM move to state 6. After factory completed, state automatically move to state 7. We simplified the FSM, cause after a simple compute this process brings very little influence to the game result, but it makes the system very complex. Some time the required factories are more than one, due to the FSM, it may building it one by one. We can't afford this waste. A trick used here to solve this problem. As Tab 9 shows, we using switch to implement the FSM. We put state 6 before stat 4 without a break. Then the state 4 would run every cycle. In this case, we could build more than one factory at once.

```

Building_check(){
    Switch (building_state)
    Case (0 :
        .....
    Case (6):  for factories
                If finish
                    Build("machine shop");
                If all factories is finished.
                    Building_States=4;
    Case (4):  calculate F_num;
                If (F_num > current_num) //current num cont ains the one in building
                    Building("factory");
                Break;
        .....

```

Tab. 9 Trick in build more buildings.

Scenario 2: decide what to craft.

What to craft is the most important decision in AI. A certain number of troops and a right setup, win becomes so easy. In this rush attack AI; time is the first thing to consider. In order to craft the troop with highest force, the expression in Fig 24 would divide the building time to gain a value that denote the force is produced per cycle. Secondly, we must using resource in a right way. Hence there is a method to

temporally reason the crafting process. This method would help AI to make a crafting plan which is most suitable with current money. At last, crafting check is loaded by each buildings, a “For” function view all buildings with crafting ability.

a) Crafting Reasoning

As Fig 24 presents, we take time in consideration. Besides time, money is an extra element. Overall, time should be first, and then money. This conception provides to maximal the force in the limited time.

First of all we describe the situation when money is enough. In this case we just craft the unit with most force per cycle. Obviously, tank has the biggest force. It is the simplest decision.

Secondly, in standard game definition, barracks only can craft marine; factories without machine shop could craft bike and those with machine shop craft tank. As we present in scenario 1, all factories would have machine shop. Hence, the situation is simplified; factories handle both bike crafting and tank crafting.

Let we introduce the barrack process first. In the beginning, compare marine with bike, the crafting time and cost is a bit higher then marine but the force is almost triple as marine (T7). Consequently, once the factory is completed, no marine crafting is needed. View the expression below, we would introduce the reasoning process.

$$\frac{\text{Force } i}{\text{C_time}} * \text{proc_time} = \text{RS_force}$$

Force i : force of one type of units.

C_time : crafting time.

Proc_time: the possible crafting time.

RS_force: a value used in comparison, craft or waiting for craft the units with higher RS_force.

Fig 27 Expression of RS_force.

This RS_force indicate the force that would gain if we continue crafting the same unit. For instance, building states still in state 2, it means factory starts to build or already in processing. Now we need to decide craft a marine first or wait for the factory completed. Use the RS_force above, proc_time for marine is from the current cycle to the cycle that the first tank crafting process finish. It is equal to remain of factory building time plus building time of machine shop and the crafting time of a tank. In

this case, if $RS_force(\text{marine}) > RS_force(\text{tank})$, a decision is made that start to craft marine.

Now we come to the building states that some factories were finished. In general, if the money is enough to craft tank, it is no doubt to crafting tank. Otherwise, we calculate RS_force , in this situation, $proc_time$ have a different way to compute (Fig 28).

$$Proc_time = \frac{250 - cur_money}{E_money} + C_time(\text{tank})$$

Cur_money: current money

E_money: money gain per cycle

C_time(tank): crafting time of tank

250: cost of a tank

Fig 28 compute $proc_time$

RS_force is the way that used in our design. A learning method in the final conflict will change the value N . the change of N cause s the change of Force expand to RS_force and the whole crafting sequence . Furthermore, the value i is calculate based on the opponent troop. Hence, in the new crafting sequence more suitable units have a certain priority. Detail presents in scenario 3.

Scenario 3: Updates adjust value N .

As we presented in Fig 25, the expression shows the way to compute the adjust value N . In game processing, this calculation would be finished after the game ends. Collections of the variables that used in Fig 25 should be done during the final conflict. The section describes the battle control strategy. Afterwards, we give the updating detail.

In the final conflict, we use the simplest attacking method: random attack. Attacking units randomly choose a target and fire till target is dead. Target changing only occurs while its target dead. Variables SHP_i and SAP_i record the total damage done by a specified units and its target. Two elements influence the sum of HP. The first one, due to the armor type (see Fig 13), armor works as shield with absorb the attack power. In this case, the HP is equal to attack power minus armor. In the second, in some situation that its target doesn't have as much HP as the damage it can cause. Hence, the lost HP is equal to its current HP; the additional Attack power is wasted. Consequently, the adjust value try to confirm that it value presents the influence of these kinds very precisely.

Recording part is simple, now it is the updating part. In order to show the learning process, we give up some efficient method. In our design, the updating uses an old and slow method. It is an average method that just sum value N in different games, and divide the total number of games. Each time when N is updated, the new value would be used in the next game. In addition, we try to make sure the value N would be universal from different opponent troops. Value N is split into smaller elements: $V(\text{type}, \text{target})$. Fig 29 presents the expression and update method.

$$V(\text{type}, \text{target}) = \frac{\text{Shp } i}{\text{Sap } i}$$

$$V_{N+1}(\text{type}, \text{target}) = \frac{V_N(\text{type}, \text{target}) * N + V(\text{type}, \text{target})}{N+1}$$

V_{N+1} : the V value in the next game.

V_N : the V value in the Nth game.

Fig 29 $V(\text{type}, \text{target})$ and update method.

4.2.3 Sum up

We re-describe our design in a general format. Then we explain our design as this form. It is an Advanced Point system based on the conception of evaluation system in commercial RTS games. We add the game states effect and the experience to make the point changeable. The point is “learning” from game to game. The more precise it is, the higher probability win the AI have.

It is a quintuple $\{P, F, S, E, M\}$. Fig 30 represents its relationship.

- P: the point of units.
- F: the unchangeable game information, like hit point, attack power and so on.
- S: game states which may influence the point. For example the set up of opponent troop
- E: Experience, it the key structure in this system, it's a probability that denotes the performance of units in real conflict. It could have different format.
- M: method to learning, in a word how to make the experience more precise and efficient.

$$P = F * \sum (E * S);$$

$$E_{\text{new}} = M (E);$$

Fig30 Relationship of expressions

The expressions used in our design is presented in Fig n. we just implement a most simple way to proof our conception. Only three units are considered and opponent troop is predicted. In addition, the structure of each expression can be more complex to show other RTS game features, these could be the future research.

$$\frac{HP * AP}{CD} * \sum \left(\frac{SHP_i}{SAP_i} * \frac{Num_i}{T_num} \right) = Force$$

\downarrow \downarrow \downarrow \downarrow
F **E** **S** **P**

Num_i: number of units i in opponent troop.
T_num: total number of units in opponent troop.

Fig 31 Example of advanced point system

The method in our design it doesn't show in the Figure. As we presented in 4.2.2 scenario 3, it is use an average method.

4.3 working with ORTS

Till now, we almost finish our design. Just one step left to implement the theory in ORTS. Unfortunately, we meet some problems in this implementation. These problems lead us to choose an eclectic implementation: simulation. In the following part, we explain these problems. Some of them have solved, some not.

4.3.1 Problem 1: lower FPS

In original design, ORTS is working in Linux. Our specified operation system is Windows. Although, a WIN32 package is designed to fix Windows environment, we still have some inconvenient.

First of all, we open ORTS project in VS 2005 edition. After compile, we get four executable files with the suffix “.exe” including the server and client. These files automatically store in the path:”orts/trunk/bin”. The server runs fine in this path, but fatal errors take place when the client runs. Fig 32 presents the error.



Fig 32 Runtime error in client

After debug several times, we can't find the solution. We forward this problem to ORTS help forum. Fortunately, we got an efficient response. Although it doesn't explain the reasons, it just tells that copy these files to path:”/orts/trunk”. The first problem is solved.

The second mini problem actually is not a real problem. We found that if we run client with 3D graphic interface. The client performed really badly. The FPS is really low, like one or two FPS. It is not work. We have done many things to improve, such as, updating the graphic support package. The performance is improved a bit. But still it is more than 8 FPS. In this case we drop the 3D graphic interface. Alternatively, a 2D graphic is used to check if the units are working in the right order.

4.3.2 Problem 2: move continuity

After solve the graphic interface, we can starts to implement our design. At the beginning steps, implementation is about the basic movements, such as move, building and attack. In this section, we describe the problem in basic move () function.

As we mentioned in above sections, compute_action () is loaded in each cycle (see section 3.2.2). Clients can set action to in game units. Based on the pervious information, we implement move () with a chosen unit, a destination and the max speed. The tutorial said that the unit would move to destination in the following frame. But, problem raised here. In the graphic interface, the unit only moves one step after the action is set. In order to make the movement continuity, we go ORTS help forum again. We find an answer that this happens in the latest snapshot. It doesn't have a real solution. However, it gives two suggestions. One is using path finding to set a predicted path to unit. The second one is using a destination variable, that the unit can

check its current position in each cycle and keep moving till it reaches. We choose the easier way: the second. There is a nice implementation of this path finding process. It is written by Asbjørn Bydal, Frode Nilsen [17].

4.3.3 Problem 3: building

Besides `move ()` function, `Build ()` is another important basic behavior in ORTS. We tried several times in our research, but it doesn't work. We also post this in help forum. Anyway, no response is available yet. This is the direct reason to why we choose simulate ORTS system and implement our design in simulation.

Build action is a function defined in `PlayerCommander` and `PlayerActions`. We tried both way, but they don't work.

Firstly, we start with `PlayerActions`. In the eventhandler, we include `PlayerAction.h`, and then create an object PA of class `PlayerActionHandler`. In this case, build function can be loaded in eventhandler class. After set up the parameters, it seems that everything is in right way now. Then we compile client program and run it. The compiling is successes; also no error happens when it runs. But nothing happens in game, work doesn't implement the build commands.

Secondly, we tried another way: `Player commander`, the beginning steps is the same, including head file and create object. The first problem take place here, the constructor needs an initial class object "ModuleSet". After satisfied the initial requirements, we could successfully create object and compile the client. A fatal error occur when the client runs.

We present the error page and the partial codes appendix A2.1. Due to the time issue; we finally implement in a simulation system instead of ORTS.

5. Implementation

In this chapter, we implement our design in a simulation system. It is a simple simulation of ORTS. It only implements the basic function which is related to our design. In a word, it is a test bed only for our design.

5.1 System overview

In general the simulation environment is written in JAVA. The game type in simulation is ORTS standard game. Due to some structure changes in simulation, the game information is kindly adjusted to fit new environment. First of all, we give a system overview of simulation system. Then, we introduce the simulation functions that the system provided. At last, it is the changes of game information.

5.1.1 Overview

This section, we present the relation of each class. Before introducing the system we would like to discuss the definition of game style.

As we know ORTS is a huge system. Our simulation would like to implement partial function that relevant to AI research in this paper. In requirement specification (section 3.3), we describe that research is focus on decision making and learning. It is a single AI play the game to craft a well troop to fight with a predicted opponent troop. The learning process located at the end of battle to change the adjust value N . Hence, the system simulates the process of AI player. AI use a rush attack strategy to craft a troop as soon as possible and do conflict with a predicted opponent troop. At last N is updated to increase the probability of win and decrease the time to attacking. The main task of AI is building, crafting and fighting, other elements of RTS would be ignored in simulation system.

System contains 5 classes. There are Worker, A_object, B_object, AIcore and Gui. The first three classes are game units. AIcore contains the AI function. The game environment is provided by Gui. Fig 33 presents their relationship.

In initial stage, AI starts with 4 workers. Then Gui provide the game cycle function: loop(). In each loop, it loads process in AIcore. After a serial decision making process, instant effect of decision is update to the game data which is stored in data. Besides these instant effects, the data also changes because of the time steps. Changes of this kind are updated by Gui. Loop is reloading till the game reach a result. Some special cycle number would be recorded in Gui, such as, time for attacking.

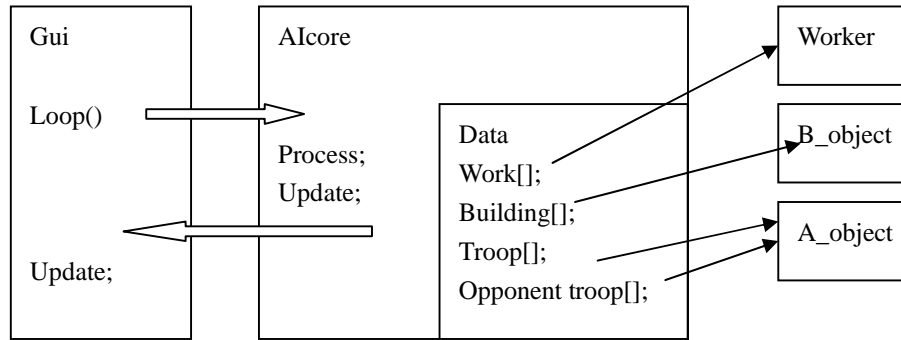


Fig 33 System overview

In ORTS, one loop is presents 1/8 seconds in real life. This brings the real-time felling to players. In simulation, we just need check if everything is in the design order and evaluation its performance. Consequently, there is no time holding method in loop (). Instead, we just assume that one cycle is one second in game world. Class Gui implements the time assumption.

In addition, data for the units in our design is based on the 1/8 second system. In order to march the simulation, we adjust some values. The information relevant to frame number is cut down to 1/8. Most values could divide exactly by 8 except cool down. After circumspect consideration, we give the cool down value of each units, marine: 2, bike:4 and tank:5 (other information refer to T5).

5.1.2 Map in three-layer architecture

In this simulation, system has 5 classes. There are Worker, A_object, B_object, AIcore and Gui. In Fig 34, we map these classes to the three-layer architecture. Also it contains the key methods in each class.

In general, classes in basic layer provide basic behaviors. Class worker provide the build () function. A_object is an abstract class for those units that can attack others. B_object denotes buildings. Three basic behaviors are build, attack and craft. However, this system doesn't contain a move behavior. View the goal, we would find the answer. Research is focus on how to make decisions to craft a powerful troop and learning. In this case, a move function would add a lot of works in design the simulation system and the AI class would so complex that we can't handle.

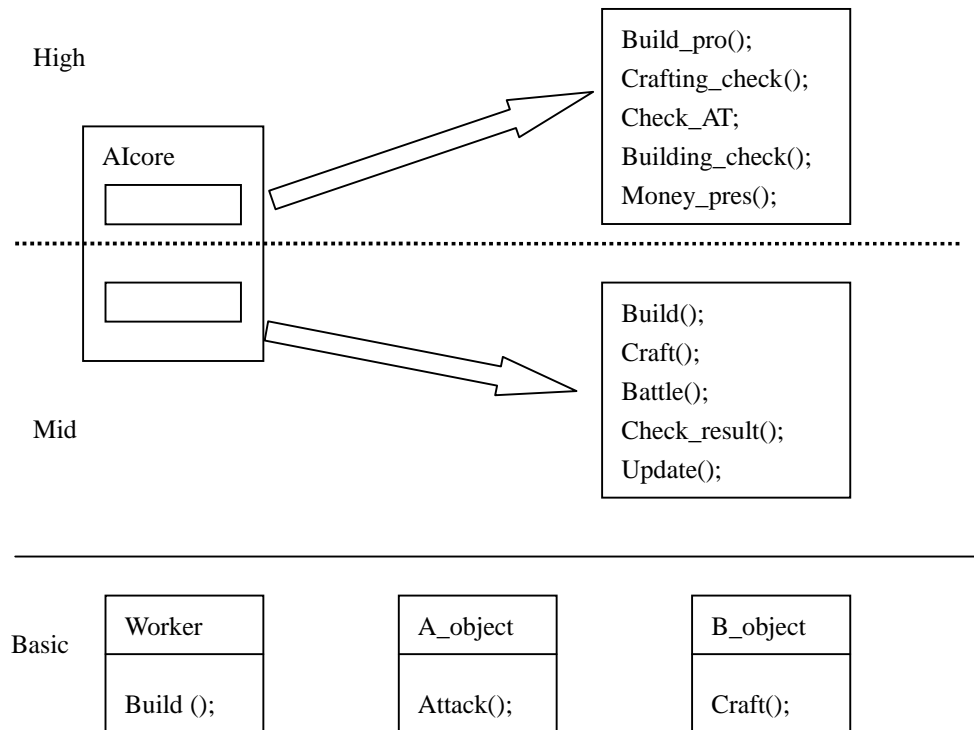


Fig 34 Mapping simulation system in three-layer architecture.

AIcore is contains functions that both in high layer and mid layer. High layer functions provide a decision making mechanism. All, general decisions are made in this layer, such as time for attacking. These general decisions forwards to mid layer. Functions in mid layer dissemble decision and implement it. The implementation would be a single behavior or a serial of behaviors.

Class Gui can't map to the three layer architecture, because it works as a server in ORTS. It provides the basic process of each cycle. For instance, AIcore is a package of car, which contains all parts; then Gui works like a assemble line, it give the right order to involve those function, and let the "car" looks like a real "car". In addition, Gui could restart game easily, it gives convenient to testing.

5.2 Implementation of each class

5.2.1 Worker

Worker is basic type in the game. Normally it handles gathering resource, building and mending task. In some special time, it would join conflict too, but there are attack power is very low. In our simulation, we redefine worker. It only needs to build buildings. Those features that don't relevant to building task are exterminated.

In our definition, work has two properties. The first is ID, which is a unique. The second is time. When a work is building, time presents the building time. Hence, while $time > 0$ means that it is occupied. Time minus one while a cycle is finished. When time reach 0, it stops decrease. A work with $time = 0$ that is free handle other building task. In addition, it implements the basic behavior build. Source code of build() see appendix A2.2.

5.2.2 A_object

This class abstracts the units which compose the attacking troop. Beside the information of units, it should contain some other variables to describe the current state of units in conflict, such as current hit point, time for next attacking. Table 10 presents all variables and short description of each .

Name(type)	description
Name(string)	Type of unit. In the constructor, name would decide other variables.
ID (int)	Unique ID.
Target(int)	It presents unit's target in conflict. If target < 0 , no target. If target > 0 , target= an ID in opponent troop.
Max_hp(int)	Maximal HP.
Cur_hp(int)	Current hp. if current hp= 0, the unit is dead.
Ap(int)	Attack power.
Cd(int)	Cool down.
Cd_time;(int)	A counter for cool down. When Cd_time remain to 0, unit could attack again.
Armor(int)	Armor value.
B_time(int)	The remained Crafting time of unit.

Table 10 variables in A_object.

In this variables description, we simplified armor type. It doesn't include those specified armor type. Because, in the definition of ORTS standard game. It has deferent type of armor, but the actually value of them are all zero. So the function shows in Fig 13 is useless. Secondly, due to the crafting process, we have B_time to presents the remained Crafting time. In the beginning of crafting, the unit is already adds to my troop array. But, this unit don't published to other functions until its $b_time = 0$.

Variables initialed in constructor. Name of the unit decide the type, each type has their unique variable value. a partial code presents how it works.

```
public A_object(String cname, int n ,int m){
    // m control the crafting side. m=1, troop m=0, op _troop
    ai=ai.getInstance();
    if (cname=="marine"){
        id=n;
        max_hp=cur_hp=40;
        ap=5;
        cd=2;
        armor=0;
        b_time=24;
        this.name = cname;
        if (m==1){
            ai.money -=50;} //AI is game stats, whom is using singleton pattern.
    }
}
```

Tab 11 Partial code of A_object constructor

There are three input parameters. The first one cname denotes the type, N presents the unique ID, and the final is M. As the comments said, M is used to distinguish the units from our troop and opponent troop. In game specification, opponent troop is predicted. Hence, a single “IF” checks the m value so that the initial of opponent troop didn’t alter the money states. Furthermore, there are two variables don’t initialed in constructor: Cd_time and target. Cd_time initial value is 0, and target is automatically set to no target :“-1”.

Beside the constructor, A_object implements an important basic behavior: attack (). This function has two input parameters and no return. One input is a new target. The other one distinguish the attack is from troop to opponent troop or inverse . At first the function would check if the unit doesn’t have a target, and then set the target N equal to the first parameter. Otherwise, it checks the current target whether is dead or not. If the current is alive, then the unit keeps this target, else it targets another one. Secondly, attack is done if the cd_time equal to zero. Then the attack power, lost hp of its target and target number is recorded. The records is updated to data in AIcore for calculate the SHPi and SAPI. If it’s an attack from opponent troop to our troop, the same process is running except the updating. (Source code see appendix, A 2.3)

5.2.3 B_object

Compare with A_object, B_object has the similar structure. It is abstract class for all buildings. First of all let’s view the variables present the relevant information of buildings. See Tab 12

Name	Description
ID (int)	Unique ID.
Name(string)	The type of building
Attachment(int)	If the building is a factory, this variable is used to present its machine shop. 1: it has. 0: it doesn't.
B_time.	Time left to finish crafting. <0, the building is not completed. =0, the building is free >0, it's in process. B_time equal to the time left.

Tab 12 variables in B_object.

The same as A_object, when the building is start to build. It already added to building array, so B_time is used to check if the building is completed. In addition, we extend B_time to present if the building is currently crafting something.

The last basic behavior is defined in this class. In section 4.1.2, we design the decision make process for each buildings. Hence, after the decision is disassembled in mid layer, and then forward to this. Nothing should be checked. The build() function is quite simple, just find the a new position in troop array and initial a A_object with the specified type.

5.2.4 Gui

The basic function of Gui is implementing the game environment. It provides the game cycle issues and the updating due to the time going. In order to presents the "learning" process; it is able to restart the game. In addition, some variables are used to denote the current game number and value N. (details see Table 13)

Name(type)	Description
Cycle(int)	Current game cycle.
Turn (int)	It presents that current game is the Nth game.
Win(Boolean)	How many games did AI win in the past N games?
Ntank, nbike, nmarie.(int)	The initial number of each unit in opponent troop
Tankn[4], biken[4], Marinen[4]	N value. tankn[0]: the adjust N for tank, tankn[1-3]present the Vi.

Tab 13 variables in Gui class

After present the variables, we introduce the functions in Gui class. In the simulation, we defined game process into two parts, the normal phase and the conflict phase. It would check the attack decision which located in AIcore at the starts of each loop during normal phase. Once the decision is made, normal phase reaches the end and simulation enters conflict phase. Due to the deferent phase, there are two kinds of loop and update functions, finally, a result check value is received from AIcore and game reach the end. Updating N is done right after the game ends. In this case, Gui runs a restart function to initial a new game. (Main loop presents in appendix A2.4)

5.2.5 AIcore.

This the core class of our system. It contains the data of this game. At first we use a table to show all these game data and a short description after each variable.

Name	Description
Money	The number of current money
Emoney	The money gain after each cycle.
Building_state	Current building state.
Worker[]	Array stores the date of workers.
Building[]	Array stores the data of buildings.
Troop[]	Array stores the data of my troop.
Op_troop[]	Array stores the data of op_troop.

Tab 14 variables in AIcore

Besides these variables, this class contains adjust value N for each unit type and variables to record damage done and lost HP which is used in calculate the SHPi and SAPI. (Resource code See appendix A2.5)

In the previous class introductions, some variables would be used in other classes. In order to void the duplicate problem, AIcore use a singleton pattern. We set the constructor as a private function so that other classes can't create this object. Then it defines a private static *INSTANCE* and a public method that returns the same instance. Different to other singleton applications, a Del () function is used so that we reset *INSTANCE*. In this case, we could reset game data for another game run. (Definition see appendix A2.6)

We sort the functions in AIcore into three kinds. The first one implements the building process. The second one is the crafting process. These two processes would be loaded in each normal loop. We call the last one battle or conflict process which implement

the conflict and the functions relevant. We would explain them one by one.

At first, it is the building process. In section 4.2.2 the first scenario, it already presented the detail of process. Hence, we would explain the process in another aspect, which presents the functions structure of it. Process starts with function `build_pro()`. It loads function `building_check()`, and it got a return string that denote the decision. if the string could match our predefined building types, `build_pro` would implement this decision. Then it calls the middle layer function `build()`. Finally, the decision is realized in class worker. Pseudo code of these functions presents in Table 15.

Class Alcore:

```

Void building_pro(){
    String = Building_check(building_states);
    If (string is match one of the predefined type)
        Build(string).
}

String building_check(int){
    Switch (building_state),
    Case(0):
    Case(4):
        Tmp=Money_pers(); // check if money states. See Fig 27.
        If (tmp>current money consume)
            Return" factory";
    .....//detail see section 4.2.2 first scenario.
    Return string;
}

Boolean Build(string){
    Choose a free worker;
    Worker.build(string); //detail see A2.2 build();
}

```

Tab 15 Functions relevant to building process.

Secondly, we introduce the crafting process in the same aspect (decision making details see section 4.2.2 second scenario). A `crafting_check` function handles this job. It checks buildings that can craft attack units. If they're not occupied, a decision is made due to the money issues (see section 4.2.2 scenario two). Then the decision is forwarded to basic layer and implement in class `B_object`. (`crafting_check()` see A2.7)

The last part is the conflict process. It is the only process in conflict phase. Game starts in normal phase. How could game enter the conflict phase? There is a function called `check_AT` is loaded in each loop during the normal phase. It checks the force of

my troop and opponent troop. Once, my troop get more force than opponent, it stops normal phase. Then the game enters conflict phase. While in conflict, function battle () realize the attacking process. It uses a random strategy for both sides. Basic attack function is implemented in A_object. The result of each attacking would be recorded by update (). In addition, at the beginning of each conflict loop, check_result is running, this returns a symbol that indicates the result of battle. Once the result show a winner, the game ends.

6. Testing

In this section, we test our implementation in three aspects: correctness, efficient and reactivity. Testing environment and data are described in pair. Tab 16 presents the initialization values of game system.

Name of variable	value
Set up of opponent troop.(tank,bike,marine)	(15.5.5)
Adjust value n for tank	1
Adjust value n for marine	1
Adjust value n for bike	1
Start money	150
Money earned per cycle	10
Assumed end cycle	600
How many games	500

Tab 16 Game initialization values

6.1 Correctness

Correctness check is aimed to test that the single game process is running correctly. The results presents below. We already cut the useless cycle information.

```

start
cycle: 1    money == 160
barracks 0
cycle: 2      money == 20
cycle: 81     money == 810
academy 1
marine    crafting.    ID: 0 610    //marine starts to craft, the id is 0, the money
                                                //left is 610
cycle: 82     money == 620    //current cycle number, and money.
cycle: 105    money == 850
marine    crafting.    ID: 1 800
cycle: 106    money == 810
cycle: 161    money == 1360
factory 2           //factory starts to build, the ID is 2.
cycle: 162    money == 1070

```

cycle: 281 money == 2160
tank crafting. ID: 2 1910
cycle: 282 money == 1920
factory 3
cycle: 283 money == 1630
factory 4
cycle: 284 money == 1340
cycle: 300 money == 1500
factory 5
cycle: 301 money == 1210
cycle: 331 money == 1510
tank crafting. ID: 3 1260
cycle: 332 money == 1270
cycle: 381 money == 1460
tank crafting. ID: 4 1210
cycle: 382 money == 1220
cycle: 402 money == 1420
tank crafting. ID: 5 1170
cycle: 403 money == 1180
tank crafting. ID: 6 930
cycle: 404 money == 940
cycle: 420 money == 1100
tank crafting. ID: 7 850
cycle: 421 money == 860
cycle: 431 money == 960
tank crafting. ID: 8 710
cycle: 432 money == 720
cycle: 452 money == 920
tank crafting. ID: 9 670
cycle: 453 money == 680
tank crafting. ID: 10 430
cycle: 454 money == 440
cycle: 470 money == 600
tank crafting. ID: 11 350
cycle: 471 money == 360
cycle: 481 money == 460
tank crafting. ID: 12 210
cycle: 482 money == 220
cycle: 502 money == 420
tank crafting. ID: 13 170
cycle: 503 money == 180
cycle: 511 money == 260
tank crafting. ID: 14 10
cycle: 512 money == 20

```

cycle: 536      money == 260
tank   crafting.   ID: 15 10
cycle: 537      money == 20
cycle: 561      money == 260
tank   crafting.   ID: 16 10
cycle: 562      money == 20
cycle: 586      money == 260
tank   crafting.   ID: 17 10
cycle: 587      money == 20
cycle: 611      money == 260
tank   crafting.   ID: 18 10
cycle: 612      money == 20
cycle: 636      money == 260
tank   crafting.   ID: 19 10
cycle: 637      money == 20
cycle: 661      money == 260
tank   crafting.   ID: 20 10
cycle: 662      money == 20
cycle: 686      money == 260
end cycle of normal phase: 686
opponent troop win
0.911329727293216  1.0  1.0

```

Tab 17 Single game process details

In Tab17, the process is very clearly. Our design is correctly running. There are some comments present the meaning of each line. One thing should be notice that the last line shows the updated adjust value N after the first game.

6.2 Efficient

This part tests the influence of learning process. It would test two game styles: original one and game with our learning method . In the following figure, we present the curve of probability of win and the end cycle. Data presented in figures prove that our design is efficient; it increases the probability and short the cycle of normal phase. In addition, Fig 37, 38, 39 shows the curve of V values for each unit.

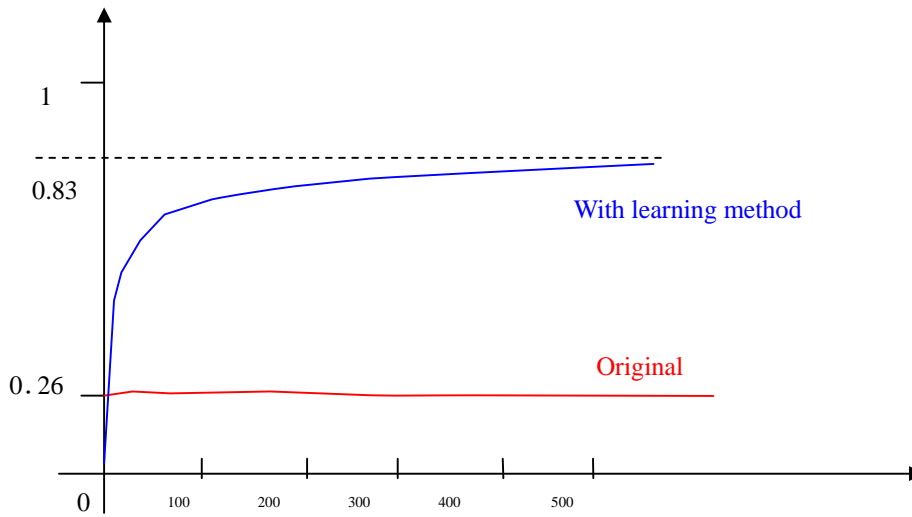


Fig 35 Probability of win

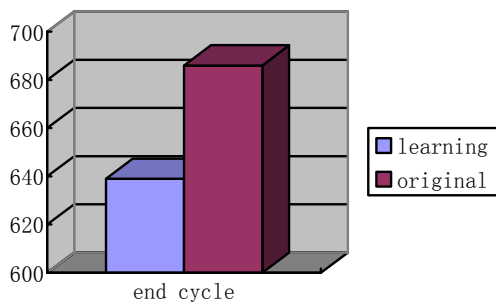


Fig 36 The end cycle of game

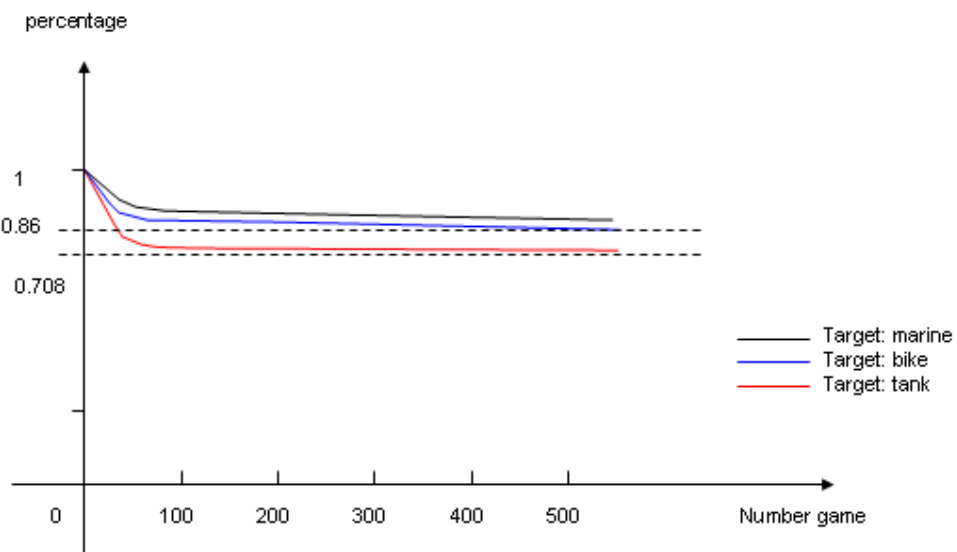


Fig 37 V value curve of tank

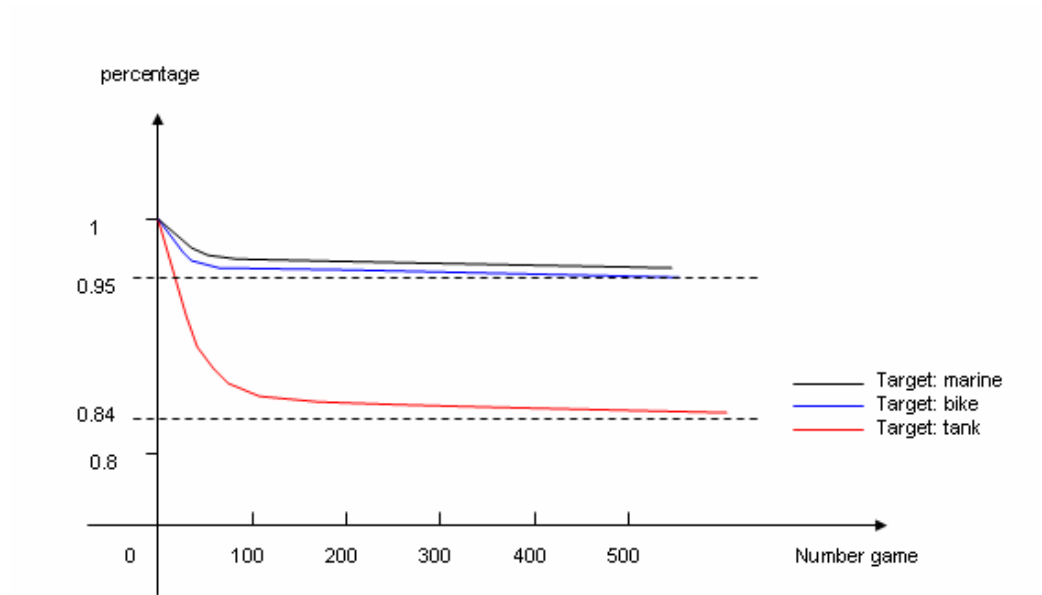


Fig 38 V value curve of bike

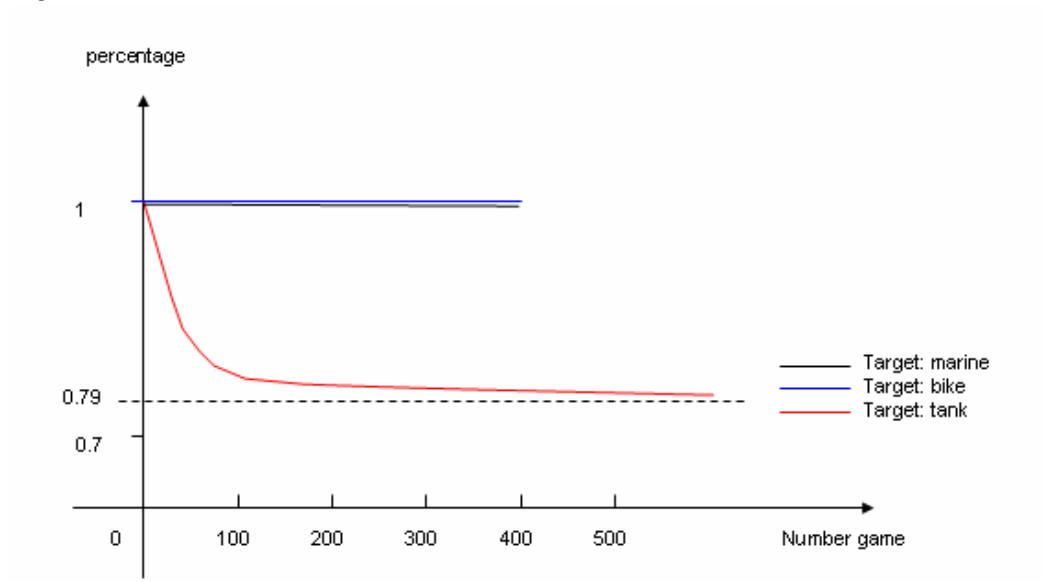


Fig 39 V value curve of marine

6.3 Reactivity

We would find the average time which the system returns to a stable stage from the moment opponent changes took place. After testing, we found that this time actually can be ignored in our research. Because, learning process gain the V value for each units. It means while the opponent troop change its set up, AI just recalculate its adjust value N with the set up information. Very little changes occur in the V value.

Fig 40 shows some sample for the V value of different troop set up.

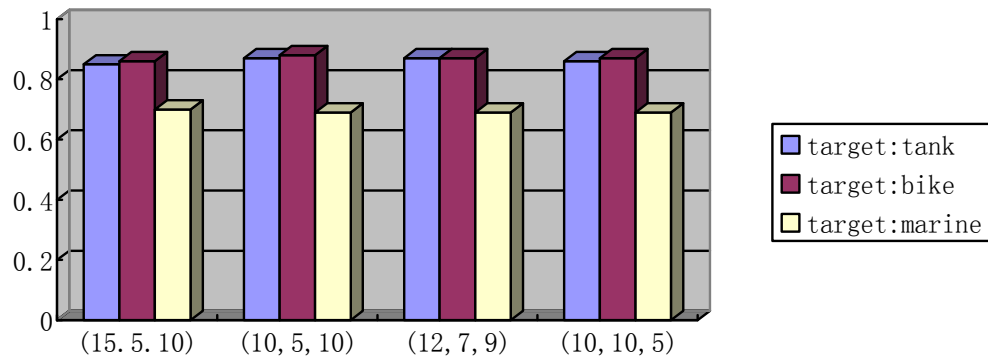


Fig 40 V value table for tank(deferent opponent set up)

We found a strange phenomenon while we test reactivity. The probability of win is increased before learning. But, while the opponent is in some special set up, the probability of win still very low. While opponent troop set up is (10, 15, 10), the win probability is increase from 0 to 10%. We admire this result and present the reason. RTS game is so changeable, but we only consider one effect element the attacking lost. It is not enough for sure. We just implement one element to prove our conception. To cover all these effect elements would be done in future works.

7. Discussion

In this section, we discuss the reasons why we design our own method instead of exist learning methods, such as learning automaton and decision tree.

First of all, based on the information that presented above, we would find that the RTS game is not stable, and the minimal cycle is 1/8 seconds. Hence, we can't independently say a decision is good or it's bad. Even, a serial of decisions may suitable for some situation. However, opponent is changes his strategy too. There isn't such an absolute powerful serial of decisions. To say the least, there isn't such a serial of decision that could always have the highest probability to win. In this case, we design our method, that each decision is made based on the current game states, and experience that gain from games as an enhancement. It maximal the benefit of each decision and lead the AI to win. Fig 41 presents an abstract conception.

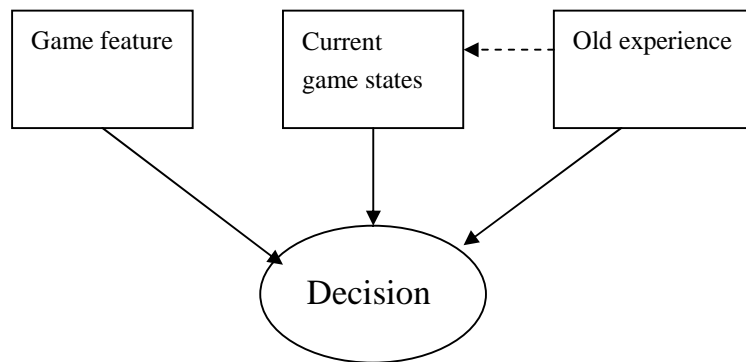


Fig 41 Features of decision making in our design

Current game state is the most important part to make decision. It computes both the states from each side, including the troop set up, number of units, snapshot of tech-tree and so on. Then this information are evaluated depend on old experience to work out a certain point. AI would choose the decision which got the highest point. A direct connection from old experience to current game states shows that old experience is an enhancement of game states, and help to calculate the point. For instance in our implement, adjust value N is a sum of products of V (type, target) and opponent set up.

Secondly, let's analysis what learning automaton can provide. In section 2.4.1, we present the basic idea. Now we try to merge it to RTS game environment. In general, the decision making is depends on the previous respon se. After certain times, we could gain the probability of rewards for each action. In this case, we could imagine after certain number of games. It meets a situation to make a decision, and then it could find some clue of the past decisions. What's more? The old experience shows the probability of rewards for such a decision. Fig 40 shows the decision in a way that

familiar with Fig 42.

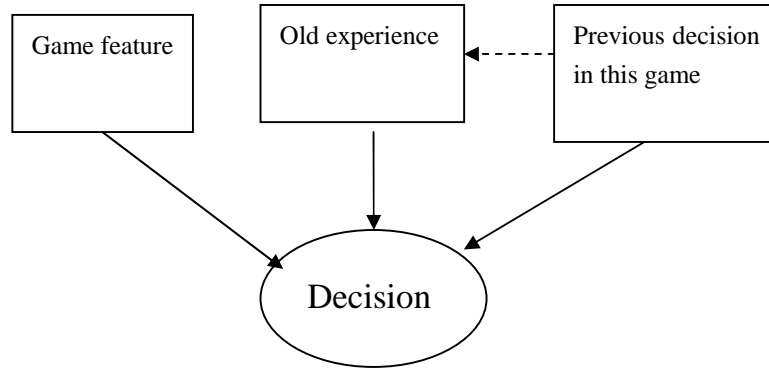


Fig 42 feature of decision making in learning automaton

It's obvious that learning automaton make its decision depends on its experience of old games. Can we trust these kinds of experience? The answer is no. first of all; the most efficient decisions could limit opponent action to gain benefit. In other sense, the decision is directly reacted of opponent actions. In learning automaton, we can't find certain method can achieve this aim. Secondly, we admire that after number of games, it could have the right and well experience for a specified game strategy. Then if changes take place in this strategy, learning automaton would choose the decision in the old order. And the opposite response is received from the game. In this case, learning automaton's "learn" from it and change those probabilities. So it could take a long time to reach a right and well decision making mechanism again. But, this process takes a long period. In other words, it is not an efficient method. We drop this method because of these two reasons.

At last, we present the reasons why don't choose decision tree. In general, decision tree is a tech that map all the possible way of decisions to reach a certain ends. Some result of the decision are predictable, some are not. Hence, learning methods is import to uncover the unpredictable parts. The learning process shows in a following way. It would starts in a random manner, and maps the decision sequence in the decision tree. After the system ran certain times, the decision tree would able to cover all possible decision sequences or major of them. Till this situation, decision tree could follow the decision tree to choose the best way and reach a good result. Unfortunately, RTS game is so huge and changeful. Although, we believe that we could uncover the decision tree in a finite time. Actually, it's not doable and not efficient.

8. Conclusion

In this research, we try to abstract a rush-attack AI mode from exist ones and then combine it with a suitable learning method. In our plan, AI would be designed as a client in ORTS. Unfortunately, due to some develop reasons; we implement it in a simulation system.

Our main focus is the learning method. After consider some exist methods, such as learning automaton and decision tree, we found that their not suitable for the complex RTS games. Finally, we investigate our own design.

It's an advanced point system that developed base on the calculation system in commercial RTS games. In this system, each units and buildings have a point which is calculated by unit information, current game states and some "experience". AI makes decisions depend on the point. It would maximal the point gain. In general, players who got higher point, who got the highest probability to win. Our implement only covers the "experience" of the lost attacking power.

Our testing simply proves the effect of the system. In general, the probability of win is increased while the learning process is going. Finally it stops at a certain value. However, we just implement a partial element that could influence the game. So in some special situation, the probability would decrease or still very bad after learning. There are other elements that influence the game result. These would be future works.

REFERENCES

- [1] Michael Buro & Timothy M. Furtak : RTS Games and Real –Time AI Research
Department of Computing Science, University of Alberta, Edmonton, AB, T 6J
2E8, Canada
- [2] Homepage of ORTS. <http://www.cs.ualberta.ca/~mburo/orts/#Overview>
- [3] RTS page in wikipedia. http://en.wikipedia.org/wiki/Real-time_strategy
- [4] Starcraft data resource package
<http://www.wfbrood.com/Soft/ShowSoft.asp?SoftID=291>
- [5] Technology tree page in wikipedia.
http://en.wikipedia.org/wiki/Technology_tree
- [6] AI homepage in wikipedia.
http://en.wikipedia.org/wiki/Artificial_intelligence#cite_note-1
- [7] Kumpati s. narendra, and M. A. L Thathachar : Learning Automata- A Survey.
- [8] Decision tree analysis. Maintools.
<http://www.mindtools.com/dectree.html>
- [9] Finite states machine
http://en.wikipedia.org/wiki/Finite_state_machine
- [10] Introduction to finite states machine IKT 505(correct system), course material
UIA, 2007
- [11] A Primer for Decision-making Professionals Rafael Olivas
http://www.projectsphinx.com/decision_trees/index.html
- [12] ORTS source package.
[13] M. Buro. ORTS: A hack-free RTS game environment. In *Proceedings of the Third International Conference on Computers and Games*, pages 156–161, 2002.
Software: <http://www.cs.ualberta.ca/~mburo/orts/orts.html>.
- [14] Johan Hagelbäck : ORTS Tutorial
- [15] Tapani Utriainen & Michael Buro: ORTS Competition: getting started.
- [16] Tank rush wikipedia.
http://en.wikipedia.org/wiki/Tank_rush
- [17] Asbjørn Bydal , Frode Nilsen: path finding for orts. UIA, web mining course
project 2007.

APPENDIX

A1 SEARCH TABLE

Figure search.

ID	Page	Description
Fig 1	8	Blueprint sample
Fig 2	10	Technology tree of terran race in Starcraft
Fig 3	12	Detail of two layer conception
Fig 4	13	AI mode
Fig 5	15	Stochastic automaton
Fig 6	16	Environment
Fig 7	16	Learning automaton
Fig 8	17	Example of Decision tree
Fig 9	18	Decision tree after calculated
Fig 10	20	Sample of FSM
Fig 11	22	BP files of standard game
Fig 12	23	Sample definition of armor and weapon
Fig 13	23	Code from weapon.bp
Fig 14	24	Example of action definition (tank anchor)
Fig 15	25	System overview of ORTS
Fig 16	26	Sample code
Fig 17	31	Three layers architecture of AI
Fig 18	33	System overview of AI
Fig 19	35	Technology tree in our game
Fig 20	35	FSM for building process
Fig 21	36	FSM for crafting process
Fig 22	38	Sample of evaluation in war craft 3
Fig 23	38	“Units” column
Fig 24	39	Expression of force
Fig 25	40	Expression of adjust value N
Fig 26	41	Partial process graphic of building check
Fig 27	43	Expression of F_num
Fig 28	44	Expression of RS_force
Fig 29	46	V (type, target) and update method.
Fig 30	47	Relationship of expressions
Fig 31	47	Example of advanced point system
Fig 32	48	Runtime error in client
Fig 33	51	System overview
Fig 34	52	Mapping simulation system in three-layer architecture
Fig 35	62	Probability of win

Fig 36	62	The end cycle of game
Fig 37	62	V value curve of tank
Fig 38	63	V value curve of bike
Fig 39	63	V value curve of marine
Fig 40	64	V value table for tank(deferent opponent set up)
Fig 41	65	Features of decision making in our design
Fig 42	66	Features of decision making in learning automaton

Table search

ID	Page	description
Tab 1	9	Feature of tank in Starcraft
Tab 2	11	ORTS characteristics
Tab 3	15	Sextuple of stochastic automaton
Tab 4	29	Comparison of marine, bike and tank
Tab 5	36	Definition of tank bike and marine
Tab 6	39	Scores table
Tab 7	40	Scores comparison (Starcraft and standard game).
Tab 8	42	Pseudo code of building check
Tab 9	43	Trick in build more buildings
Tab 10	53	Variables in A_object
Tab 11	54	Partial code of A_object constructor
Tab 12	55	Variables in B_object
Tab 13	55	Variables in GUI class
Tab 14	56	Variables in AIcore
Tab 15	57	Functions relevant to building process
Tab 16	59	Game initialization value
Tab 17	61	Single game process details

A2 CODE RESOURCE

A2.1 error page and codes

```

GameStateModule &gsm2 = *(state.gsm);

Watcher::WatcherModule wm(gsm2)
Movement::Module::ptr mm(Movement::MakeModule(gsm2, 110));
mm->addPathfinder("Default", Movement::MakeTriangulationPathfinder());
mm->addPathExecutor("Default", Movement::MakeMultiFollowExecutor());

ModuleSet ms2;
ms2.set_game_state(&gsm2);
ms2.set_watcher(&wm);

```

```

ms2.set_movement(mm);

FORALL (objs, it) {
GameObj * gob = (*it)->get_GameObj();
if (gob && gob->sod.in_game && *gob->sod.x >= 0 && !gob->is_dead()) {
    string objtyp= gob->bp_name();
    PlayerInfo& pi = game.get_cplayer_info();
    uint4 objId = pi.get_id(gob);
    if (objtyp=="worker"){
        if (control==0){ // control is used that only work is seted with this job.
            sint4 x = *gob->sod.x -4 ;
            sint4 y = *gob->sod.y -4 ;
            cout<<"building"<<endl;
            PlayerCommander PA( ms2);
            PA.build(gob,"controlCenter", x,y);
            control = 1;
        }
        else
            break;
    }
}

```

Error : no option found.

A2.2 build()

```

//building task
public void build(String bname){
    int i;
    Aicore ai=null;
    ai=ai.getInstance();
    if (bname=="barracks"){
        if (ai.money>150){
            for (i=0;ai.building[i]!=null;i++);
            ai.building[i]=new B_object("barracks", i);
            this.time=80;
            ai.money=ai.money-150;
        }
    }
    if (bname=="academy"){
        if (ai.money>150){
            for (i=0;ai.building[i]!=null;i++);
            ai.building[i]=new B_object("academy", i);
            this.time=80;
            ai.money=ai.money-150;
        }
    }
}

```

```

    }
    if (bname=="factory"){
        if (ai.money>300){
            for (i=0;ai.building[i]!=null;i++){
                ai.building[i]=new B_object("factory", i);
                this.time=80;
                ai.money=ai.money-300;
            }
        }
        if (bname=="attachment"){
            int n=0;
            i=0;
            if(ai.money>100){
                for (;ai.building[i]!=null;i++){
                    if(ai.building[i].name=="factory"&&ai.building[i].attachment==0&&ai.building[i].b_time==0){
                        ai.building[i].attachment=1;
                        ai.building[i].b_time=-40;
                        ai.money=ai.money-100;
                        // System.out.println("attachment craft");
                        break;
                    }
                }
            }
        }
    }
}

```

A2.3 attack();

```

public void attack(int n,int m){
    ai=ai.getInstance();
    int dmg=0, hp=0;
    if (this.target<0){
        this.target=n;
    }
    if (m==1){
        if (ai.op_troop[n].cur_hp>0&&ai.op_troop[this.target].cur_hp<=0){
            this.target=n;
        }
        if(ai.op_troop[this.target].cur_hp>0){
            if (this.cd_time==0){
                this.cd_time=cd;
                dmg= this.ap-ai.op_troop[this.target].armor;
                if (ai.op_troop[this.target].cur_hp>dmg){

```



```

n=ai.check_result(); //n=2 the conflict is still runing. n=1 my troop win, n=0
if (n==1) {          //opponent win
    System.out.println("my troop win");
// update_n();
    win++;
}
if (n==0) {
    System.out.println("opponent troop win");
}
//System.out.println(ai.biked2+" "+ai.bikel2);
update_n();
cyc=0;
}

```

A2.5 data definition for adjust value N and relevant variable.

```

double tank_n=0.6; //initial value of N, if it doesn't set N.
double bike_n=0.9;
double marine_n=0.9;
double tankp=30*150/5; // the unchanged part in force calculate.
double bikep=20*80/4;
double marinep=5*40/2;
double tankd1=0,tankd2=0,tankd3=0; //1 to marine 2 to bike 3 to tank.
double biked1=0,biked2=0,biked3=0;
double marined1=0,marined2=0,marined3=0;
double tankl1=0,tankl2=0,tankl3=0; //1 to marine 2 to bike 3 to tank.
double bikell1=0,bikel2=0,bikel3=0;
double marinell1=0,marinel2=0,marinel3=0;

```

A2.6 singleton definition in Aicore

```

//define Aicore as a singleton pattern
private static Aicore INSTANCE= null;
private Aicore() {
    .....
}
public static Aicore getInstance() {
    if (INSTANCE==null) {
        INSTANCE= new Aicore();
    }
    return INSTANCE;
}
public static void del() {
    INSTANCE=null;
}
}

```

A2.7 craft_check()

```

public void craft_check() {
    int i=0;
    double tmp, tmp1=1;
    for (;this.building[i]!=null;i++);
    i=i-1;
    for (;i>=0;i--){
        if (this.building[i].name=="factory"&&this.building[i].b_time==0) {
            if (this.money>250) {
                this.building[i].craft("tank");
                //System.out.println("tank craft");
            }
            else {
                if (this.money>70) {
                    tmp=(250-this.money)/this.emoney;
                    tmp1=tankp*tank_n/(50+tmp);
                    tmp=bikep*bike_n/30;
                    if (tmp>tmp1) {
                        this.building[i].craft("bike");
                    }
                }
            }
        }
        if (this.building[i].name=="barracks"&&this.building[i].b_time==0) {
            if (this.build_state==1) {
                tmp1=-this.building[1].b_time+80+40+50;
            }
            if (this.build_state==2) {
                tmp1=-this.building[2].b_time+40+50;
            }
            if (this.build_state==3) {
                tmp1=-this.building[2].b_time+50;
            }
            if (this.build_state>=4) {
                tmp1=50;
            }
            tmp=tank_n*tankp/tmp1;
            tmp1=marinep*marine_n/24;
            if (tmp1>tmp) {
                this.building[i].craft("marine");
                //System.out.println("marine is crafting");
            }
        }
    }
}

```