USING MONTE-CARLO PLANNING FOR MICRO-MANAGEMENT IN STARCRAFT

Wang Zhe, Kien Quang Nguyen, Ruck Thawonmas, and Frank Rinaldo
Intelligent Computer Entertainment Laboratory
Ritsumeikan University

I. ABSTRACT

This paper presents an application of Monte-Carlo planning for controlling units in the game of StarCraft. We developed an original simulator for applying MCPlan to solve the problem of random and simultaneous moves in a RTS game. We also apply an $\varepsilon - greedy$ algorithm to model the opponent in a simulation for improving MCPlan's performance. The experiment result is provided at the end of this paper, which shows a potential of MCPlan in this domain.

II. INTRODUCTION

StarCraft is one of the most popular RTS game developed by Blizzard Entertainment. The diversification and extremely balanced gameplay not only provides players with multiple options, but also enables it to be an ideal platform to test different AI approaches. So far, although some work has been done on building human-level AI for StarCraft, because of the real-time properties and large unpredictable random, it is still a challenging game for AI research [1].

The main issue in this paper focuses on the micro management of StarCraft gameplay. Micro-management is series of actions that issue commands to each unit of a certain group for maximizing their effectiveness during combat. Good micro management is a significant part of RTS game in which it can bring player advantages in the game, even change result of a whole match.

Our goal is to build expert AI for micro-management. In our case, we apply both a rule-based approach and a Monte-Carlo Search to control specified units. Generally speaking, we use MCPlan for decision making, however, in order to enable AI perform high-level action, we define some candidate plans using a rule-based approach. During the simulation section, we apply different ways to control each side units' group. On one hand, we adopt Top-Level Search for our each individual unit, namely, predefine several rule-based plans and reward function, then run those predefined plans as much as possible and always chooses the combination that returns the highest reward. On the other hand, an ε -greedy method is adopted for enemy units in order to model opponent actions.

Monte Carlo simulation has been applied successfully in RTS games for the strategy aspect. Such as, MCPlan can be used for high-level planning selection [2], or as a solution for large sequential decision making problems by combining with non-linear VFA (value function approximation) and text recognition technique [3]. Also UCT (upper confidence bound) for tactical assault planning has been used in RTS game for tactical level strategy [4].

Previous works on micro-management include adopting multi-agent in StarCraft [5] and a Bayesian model for RTS control [6]. However, MC simulation is rarely found in low-level AI module like micro-management.

MC simulation has the advantage of avoiding too much

expert knowledge to determine a result and it has a potential to **select** the best choice. We try to apply this technique on this problem and test how good it performs.

The contributions of this paper are as follows:

- Implementation of the MCPlan with expert knowledge for micro-management in StarCraft.
- Original simulator for complex commercial RTS game combat scenario and a detailed characterization of its design.

In section 3 we discuss our methodology, including algorithm we adopted and the design of the simulator. Then in section 4 we describe the implementation of MCPlan in StarCraft, focusing on related parameters that can influence its performance. Also we provide experiment results, conclusion and a plan of future work in section 5, 6 respectively.

III. METHODOLOGY

Monte Carlo Planning

MCPlan is a mechanism that is based on simulation and does a stochastic sample of possible choices. It determines the best statistical result after multiple roll-outs. It is an effective method to handle random and imperfect information problems with alternating moves, such as chess, poker. A great advantage of MCPlan is the reduction of expert knowledge required, instead of defining every detail through expert knowledge, MCPlan relies on effective evaluation function.

Search Algorithm

The basic view of our MCPlan is as follows:

1) Loading predefined rule-based action sets (plan) one by one to simulator.
2) Simulate each plan and evaluate it.
3) Choose the plan whose reward is highest from all candidate plans, then simulate and reward it again.
4) According to different reward value from each roll-out to adjust each weight and recalculate the reward of the plan.
5) Repeat step 3 and step 4 as much as possible.
6) Choose the plan with best statistical result for the AI player in a real game.

And here's our algorithm for searching for the best result, namely, UCB1 algorithm, showed in Figure 1 [7].

$$UCB1(i) = R_i + C\sqrt{\frac{\ln N}{N_i}} \qquad (1)$$

In the formula, i is index of each plan, R_i presents the reward that plan i obtained from reward function. C is a predefined constant, N and N_i are the overall number of run times and number of times that plan i has been visited respectively. In this formula, the first part R_i can be regarded as an element of exploitation because of the repeat trial. In contrast, the second part $C\sqrt{\frac{\ln N}{N_i}}$ is an element of exploration due to the encouragement of trying other possible solution over time. For example, in step 3 and step 4, the previous best choice may be replaced by a new choice if it doesn't perform well after several times roll-out.

Since we can find the best choice in limited simulation time by adopting UCB1 algorithm, which means the problem then turn to accuracy of the simulator and the performance of the reward function.

Evaluation function

At the end of each roll-out, an evaluation function is called to judge how the plan performed. This function should give a heuristic value and provides a positive effect for the search algorithm. In linear Monte-Carlo search, it usually consist of feature R and vector parameter ω , and has a linear relationship with R' where $R' = \omega \cdot R$. The parameter can be either fixed by

hand or updated via gradient descent.

Simulator Design

Unlike high-level strategy, high-level abstraction is not available for a micro-management simulator. Since micro-management is a much lower level action with perfect information, and it is affected greatly by many small details. Thus our simulator has to emulate real-game as much as possible.

Generally speaking, our simulator can be decomposed into five parts as follows:

A) Character Modeling

In this part, we try to create characters with every detail that is exactly the same as in a real game, such as character's hit-points, attack range, special skills et cetera. For example, each type of character can be defined as unique class with different variables. Each individual character's behavior should be defined in detail, such as how unit moves, and how unit attacks.

B) Map Modeling

Map Modeling is an essential part for the simulator because of its irreplaceable importance in RTS game. Map modeling should consider of different terrains, characters' location, and unit overlapping problem. Besides, it can also be used for denote a game state. For example, in Capture the Flag game, map is divided into tiles where units are located in. Different distribution of units on the map can be regarded as an abstract way to show different game states.

C) Candidate Plans

As a essential part of MCPlan, the quality of plan affects simulation result significantly. This part has to rely on expert knowledge, a plan can be simple as only one single action, or can be as complex as a series of detail covered actions. One possible way is to give some high-level actions and allowing MCPlan to combine them in random ways, so that AI can exhibit creative behavior [2].

D) Reward Function

At the end of each move sequence, a reward function is needed to evaluate each plan that is being tried. We designed a reward function from expert knowledge that includes all important features of game, and each feature multiply by a fixed weight ω_i , which indicates the importance of each feature.

E) Simulator Roll-outs

Before running a simulation, there is some preparation work to do, such as mapping the rule sets to each unit, Initializing parameters which will affect MCPlan by coping current game state. Essentially, each roll-out should be limited in certain steps[1]. Current game state will be checked before roll-out as well as after roll-out and reward function should be called for evaluation at the end of this step.

IV. APPLICATION TO STARCRAFT

We apply our MCPlan algorithm to micro-management part of the game Starcraft[2].

StarCraft is a multi-player RTS game in which player commands an army against with opponents. The reason why we choose this game as a platform is because of its well-balance and highly-operability, which provides us various scenarios to test our method.

BWAPI

The Brood War Application Programming Interface (BWAPI) is a free and open source C++ framework for creating AI modules for StarCraft: Broodwar. Using BWAPI, programmers can retrieve information on players

[1]In our case, each simulation roll-out means one individual unit finish executing and rewarding one plan.

[2]StarCraft and its expansion StarCraft: Brood War were developed by Blizzard Entertainment™.

and individual units in StarCraft as well as issue a wide variety of commands to units. For more information about BWAPI, we refer readers to [8].

StarCraft Combat Simulation

Character Initialization BWAPI provides hooks into StarCraft and enables the development of custom AI for StarCraft, which enables us to extract units' information from current game state, such as unit type, hit-points, and unit position. So that we can copy the whole current game state to the simulator and finish character modeling.

***Map Representation*:** In the map representation, on one hand, we break the map into tiles that are proportional to a real game map, so that units' exact game coordinates can be set in a proper way into tiles. On the other hand, we adopt empty terrain for whole map in order to keep it simple and obvious.

In addition, we used flags to denote the tiles that are already being occupied by units to prevent unit overlapping.

Plan Definition: We try to avoid too complicated a plan in order to give MCPlan enough space for search. But rather than randomly taking simple actions, we mix some rule-based plans in candidate list. For example, a plan might be like this: if unit's hit-points below 20%, then move back. In this way, we not only improve effectiveness but also make sure the efficiency of plans.

Reward Function: Our reward function is designed base on individual unit. It is suitable for both our side and enemy side. Although there are lots of elements affect game state, basically our reward function contains 4 aspects below:

- *Hit-points*: Including current individual HP of our side and enemy total HP. On one hand we should avoid HP reduction, on the other hand, more attack than move is encouraged. In this case, we represent unit's attack frequency by considering the difference of enemies' total HP during each roll-out process.
- *Energy*: Some units have a capability to use special skills during combat. Obviously, the more skills use the better. So similar to enemies' HP, we consider the difference of this value during each roll-out.
- *Distance*: Distance also is an important element that should be taken into account. Because for each individual unit, it should always stay close to allies and keep from being isolated or being surrounded by the enemy. Therefore, we calculate the difference of distance between closest ally and enemy center for each individual to judge different reward.
- Plans: We also take plans into consideration as a part of reward function. Same as other game features, each plan has a unique weight too.

We fixed all the weights from expert knowledge. With the same reward function, our units use the UCB1 algorithm whereas enemy units use the $\varepsilon - greedy$ algorithm to calculate reward value and make final decisions.

Simulator roll-outs: During the simulation process, one roll-out is limited in 20 steps. At each step except current controlling unit, we assume other units move randomly. Besides, in order to maximize the potential of MCPlan, the movement of opponent military is set prior than ours.

V. EXPERIMENTS

We tested our method on StarCraft game play [9]. Experiments were run on PCs with 3.20GHz CPU and 4G RAM.

Map

In order to make sure all units get involved in combat as soon as possible, we create a small map with all plain terrain and limit combat space. That is, two forces start in a short distance but not within attack range at the

beginning of game. The map is shown in Figure 2.

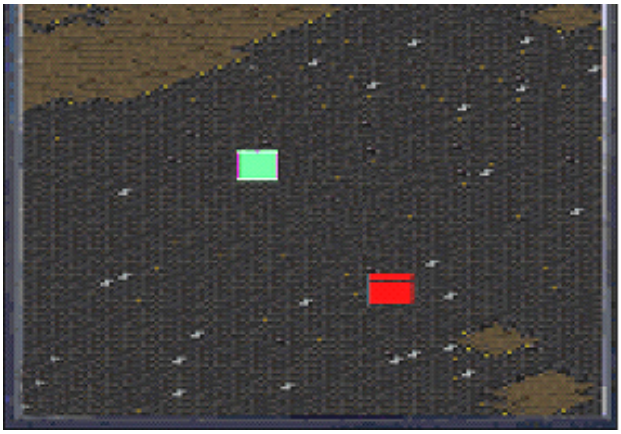

Figure 2 : Game map for experiments. The green and red areas on map are start locations of our AI and enemy respectively.

Combat Scenario Design

In the experiment, our AI fights against the original AI of StarCraft, mission objective is to kill all units of the opponent for a victory. We designed 3 experiments of Terran versus Zerg in StarCraft. The technique for Terran are U-238 Shells, Stim packs and healing, whereas for Zerg is Metabolic Boost. For the detail of units and skills, we refer reader to [9]. Figure 3 shows a screen shot of combat :

Figure 3: A screenshot of combat between Terrain(bule) and Zerg(green).

***Experiment 1*:** Four Terran Marines against six Zerg Zerglings.

***Experiment 2*:** Three Terran Marines aginst 1 Zerg Lurker.

***Experiment 3*:** Five Terran Marines and one Terran medic aginst six Zerg Zergling and 1 Zerg Lurker.

We run each experiment for 20 times and do same experiment on original AI in StarCraft. Then compare the performance.

Experiment Result

The experiment result is shown in Table1:

	Our AI Wins(%)	Original AI of SC Wins(%)
Exp1	30%	About 2%
Exp2	33%	About 3%
Exp3	40%	About 3%

Table1: Experiment Result

The result above shows the wins rate of our AI and Original AI of StarCraft in same experiments. It is obvious that original AI played extremely bad and our AI performed not ideal enough.

One possible reason is may be less run time of simulation and less accuracy of game process modeling.

But MCPlan still showed its potential because of largely increase the wins rate compare with original game AI.

VI. CONCLUSION AND FUTURE WORK

This paper has presented a preliminary work on solving the problem of micro-management in RTS games. We have described a mechanism of applying MCPlan to the game Starcraft. After we analyzed and identified the domain, we successfully developed an original simulator for the game. Our work includes plans design, game process simulation and reward function creation. From the

experiments, we got initial result which indicated a promising potential of MCPlan in this domain, but still not ideal enough.

For future work, we intend to improve simulator which requires more accurate game simulation and more efficient reward function. For example, a better abstraction method or the weights updating by gradient decent may greatly strengthen the result of MCPlan.

VII. REFERENCES

[1] B. Weber, M. Mateas and A. Jhala, "Building Human-Level AI for Real-Time Strategy Games", *AAAI Fall Symposium on Advances in Cognitive Systems (ACS 2011).*

[2] M. Chung, M. Buro, and J. Schaeffer, "Monte carlo planning in RTS games," in *IEEE Symposium on Computational Intelligence and Games (CIG)*, 2005.

[3] S.R.K Branavan, David Silver, Regina Barzilay, "Non-Linear Monte-Carlo Search in Civilization II",*in Proceedings of IJCAI*, 2011.

[4] R.-K. Balla and A. Fern, "Uct for tactical assault planning in realtime strategy games," in *International Joint Conference of Artificial Intelligence*, IJCAI. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2009, pp. 40–45.

[5] Alberto Uriarte Pérez, "Multi-Reactive Planning for Real-Time Strategy Games",http://nova.wolfwork.com/files/Multi-Reactive_Planning_for_Real-Time_Strategy_Games.pdf

[6] Synnaeve, G. and Bessiere, P. "A Bayesian model for RTS units control applied to StarCraft," in *IEEE Conference on Computational Intelligence and Games (CIG)*, 2011

[7] http://lane.compbio.cmu.edu/courses/slides_ucb.pdf

[8] http://code.google.com/p/bwapi/

[9] http://sea.blizzard.com/en-sg/games/sc/