チーズ学習：遺伝的アルゴリズムを用いたStarcraft初期オーダー創造

ミラー ミッチェル[1,a)]　アランニャ クラウス[1,b)]

概要：本研究では、進化的計算を用いた確率論的ゲームのためのコントローラの最適化を探究する。具体的には、提案コントローラは StarCraft：Brood War の早期攻撃戦略（タイミングプッシュ）を最適化する。StarCraft コントローラを作成するため、C++の API である BWAPI を使用する。C++のコントローラは R 言語で実装した遺伝的アルゴリズムと通信し、ビルドオーダーを受け取り、ゲームのスコアを報告する。R 言語の遺伝的アルゴリズムは潜在的なソリューションを生成し、そのソリューションを評価し、過去のソリューションの成功に基づいて新しいソリューションを生成する。実験データを使用して、ゲーム内のさまざまな要素を評価し、StarCraft のデフォルトの AI とのゲームに勝ち抜く最良の解決策を作り出すのに、各要素がどれだけの効果をもたらしたかを確認した。これらの要素をランダムコントローラと比較し、どの要素がどれぐらい最適なタイムプッシュを達成できるかを説明する。

Learning to Cheese: Using Genetic Algorithms to Generate Build Orders in StarCraft

Mitchell Miller[1,a)]　Claus Aranha[1,b)]

Abstract: The purpose of this research is to explore optimizing a controller for a complex, stochastic game using evolutionary computation. Specifically, we use a Genetic Algorithm to optimize the early attack strategy (timing push) of an AI that plays StarCraft: Brood War. First, we built a simple AI and its build order interpreter in C++ using BWAPI, an API allowing for the easy creation of bots that play StarCraft. The C++ controller code communicates with the actual Genetic Algorithm, written in R, to receive a build order and report the scores of each game it played for evaluation. The R script orchestrates the Genetic Algorithm, generating potential solutions, evaluating those solutions, then generating new solutions based on the success of the previous ones. While collecting data, we evaluated different factors from within the game to see how much of an effect they had in helping create the best solution, one where the AI would win its games against the default AI of StarCraft. We compare these factors against a complete random search as the control, and discuss which factors contributed, more or less, to achieving well-timed pushes and build orders.

1. Introduction

Video games are complex. The methods used to win can be vast and varying. StarCraft, a real time strategy game, pits two players against each other where they gather resources, build structures and train units, in the effort to destroy their opponent's base. Such a stochastic problem like trying to win a StarCraft match is difficult to optimize. How do you devise and optimize a controller that knows how to win? We decided to learn how to program a StarCraft bot, and use evolutionary computation to teach my bot how to win.

By teaching our StarCraft bot to win, we discovered just how complex the relationship between parameters and the outcome of our genetic algorithm was.

[1] 筑波大学情報科学類
Tsukuba University, College of Information Sciences
[a)] mitchkm12@gmail.com
[b)] caranha@cs.tsukuba.ac.jp

表 1　ビルドオーダーのコマンド
Table 1　Build Order Instructions

Instruction	Description	Integer Value
Train SCV	A unit that collects resources and builds buildings	0
Train Marine	A basic offensive unit used to attack	1
Build Supply Depot	Building that increases the maximum units a player can have	2
Build Barracks	Building that allows the player to train marines	3
Send Marines	Sends idle marines to attack the enemy	4

2. Related Works

Using video games as a platform for the research of artificial intelligence has recently attracted a lot of attention. In this trend, many AI in games competitions, such as CIG [4] have provided an space to explore different approaches to this idea. Togelius et al. [5] have recently used multi objective evolutionary algorithms for generating a general starcraft agent. In this work, we focus only on the initial moves of the game. The usage of games as a playground for artificial intelligence is vast, and a lot has been learned from it.

3. Experiment

3.1 StarCraft Bot (C++)

To be able to teach a StarCraft bot, we first wrote a bot for StarCraft using BWAPI, a C++ API for StarCraft: Brood War [2]. With this API, we set up a StarCraft bot which executes build orders. A build order(BO) is a list of actions done by the Starcraft player, our bot, in a specific order. Playing as the Terran race, the bot executes build orders made up of just 5 commands, represented numerically according to Table 1.

With this, the bot had the basic building blocks to execute early attack strategies by sending offensive units (marines) to attack its opponent quickly. The bot ' s code also managed mining resources and assigning units to build buildings outside of the build order to minimize repetitive instructions being included in the build order. The StarCraft bot 's code was compiled into a dll file and placed in the game ' s directory. Using Chaoslauncher, StarCraft was launched with BWAPI and the dll file was found and run when a match started [3].

1 4 4 0 3 4 1 4 4 0 2 3 4 0 2 0 0 4 4 2 2 3 4 0 4 3 1 0 2
1 0 1 3 0 4 4 1 3 0 4 1 2 1...

図 1　ビルドオーダーの例
Fig. 1　Example Build Order

3.2 Genetic Algorithm (R)

An R script was used to create the genetic algorithm that optimizes the bot ' s build order. Using sockets, the R script send a build order to the bot and, after the match was done, the bot sends the results of the match back. After that the game was reset, and the bot waited to receive the next build order for the next match. The score results (Table 2) were used to evaluate the success of a build order and used to calculate its fitness. The R package ' GA ' was used as a base point to create the genetic algorithm for this experiment. Since build orders can vary in length, some of the package ' s default functions were changed to support lists of differing size.

表 2　ゲームクライアント出力の例
Table 2　Example of metrics output by the StarCraft client

Metric Name	Value	Metric Name	Value
Win	0	Building Score	800
Total Minerals	2321	Kill Score	200
Total Gas	0	Razing Score	150
Custom Score	0	Elapsed Time	652
Unit Score	1100		

3.3 Search Algorithm Differences

Three different versions of the R script algorithm were used in this experiment. One algorithm was a complete random search used as a control against two genetic algorithm searches. The two genetic algorithms differed only by their crossover function. The genetic algorithms will be referred to as algorithm A and algorithm B. The differences between all three can be seen in Table 3.

Almost everything was held the same for the two genetic algorithms except their crossover function. Algorithm B does a single point crossover with only a 0.5 probability. The other half of the time, it swaps the front halves of the parent build orders with the opposite parents back half in an effort to get more differing build orders. All algorithms build orders were evaluated with the fitness function in Table 3. The scores used should indicate how

表 3 アルゴリズム A と B の詳細
Table 3 Breakdown of the differences of each algorithm

Algorithm	Random Search	Genetic Algorithm A	Genetic Algorithm B
Generated Build Orders	900	900	900
Generations	NA	18	18
Population Size	NA	50	50
Selection Method	NA	Tournament (K=5)	Tournament (K=5)
Crossover Method	NA	Single Point	Half the time Single Point + Half the time swap front and back
Mutation	NA	None	None
Fitness Function	(Build Score + Unit Score + Razing Score + Kill Score) / Elapsed Time		

many units and buildings our bot constructed as well as how many enemy units and buildings it attacked. To promote attacking quickly those four scores were divided by the elapsed time of the match. Tournament selection was used with a selection pool of five because it was a tenth of the population.

3.4 Simulating Matches

Five matches were simulated per build order to reduce the effect of the default enemy bot ' s random strategies. The scores from 5 games were averaged and then used to determine the fitness of a build order. The opponent always played the race, Protoss, while our bot played the race, Terran, to eliminate the difference in strategies between races. Three simulations of 4500 (five times 900 build orders) were run, once for each algorithm. After the simulations, 900 data points per algorithm were collected. They contained all the metrics collected from StarCraft, as seen in Table 2 plus the build order ' s fitness score, its generation for the two genetic algorithms.

4. Results

This boxplot compares the fitness value of all the build orders from each search. It indicates Algorithm A was the best at creating build orders with a high fitness. As expected, the random search was the worse. This makes sense given that our random search did not base any build orders on previous build order with high fitness. Ultimately, build orders should win games. Figure 3 compares the win rate of build orders against their fitness.

Surprisingly, algorithm A did not produce many winning build orders. A correlation between fitness and win rate could not be shown for algorithm A, even though it appeared for both algorithm B and the random search. Totaling the amount of build orders that won at least 1 of its 5 matches, algorithm B has 123 build orders as compared to algorithm A with only 11.

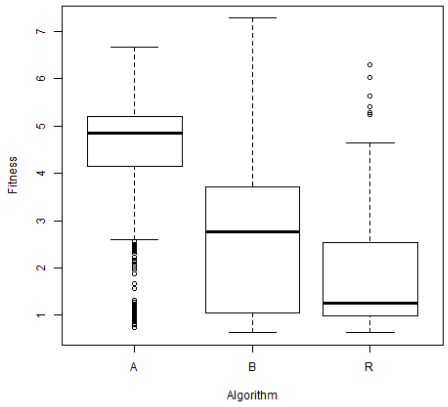

図 2 三つのアルゴリズムにより作成された回答の適応度
Fig. 2 Distribution of fitness values of all build orders generated by each of the three algorithms

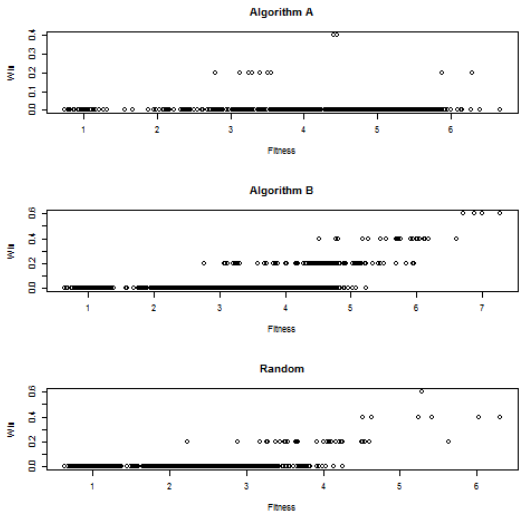

図 3 三つのアルゴリズムの適応度と勝利率
Fig. 3 Fitness versus the win rate for each algorithm

To compare the two versions of our genetic algorithm further, figure 4 shows the change in fitness as the algorithm progresses. Looking at the fitness per generation of each algorithm, A can be seen to converge.

The convergence can also be seen when looking at the

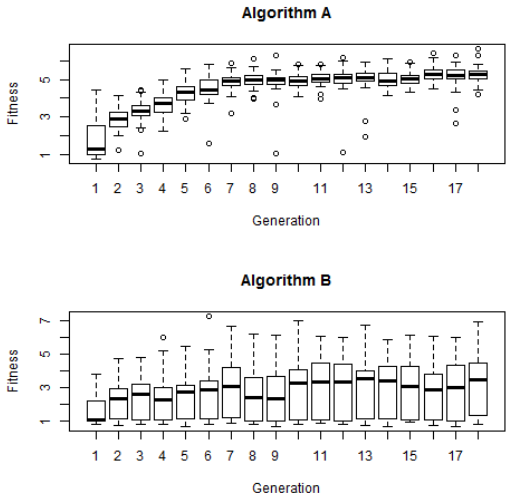

図 4 世代ごとの適応度変化

Fig. 4 Fitness values over the generations for algorithms A and B

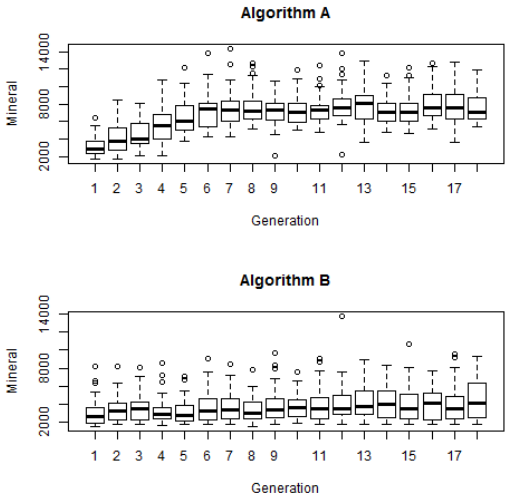

図 6 世代ごとの「リソーススコア」の比較

Fig. 6 "Resources" Score per generation between Algorithm A and B

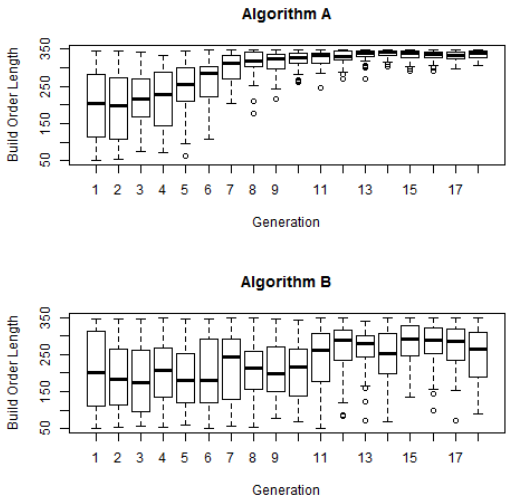

図 5 世代ごとの回答長さの比較

Fig. 5 Comparison of BO length per generation between algorithms A and B

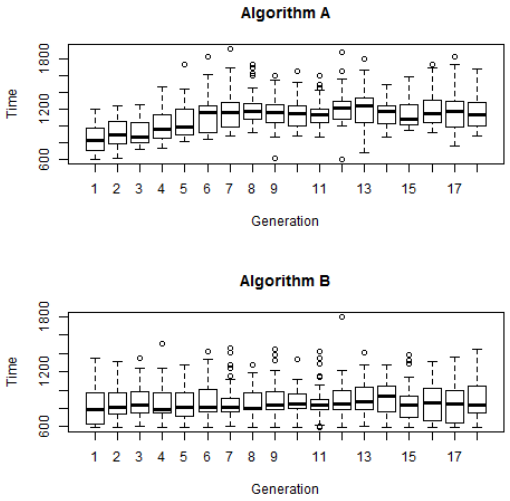

図 7 世代ごとの試合時間の比較

Fig. 7 Comparisons of the match time of all build orders per generation between Algorithm A and B

build order lengths (Figure 5). Algorithm B maintains a more spread out population in terms of fitness and build order length.

Looking at the minerals, resources, produced by each build order, algorithm A 's population 's average minerals mined increased greatly, whereas algorithm B ' s average stayed relatively stationary. It helps to look at the game time, as game length would affect the amount of minerals acquired (Figure 7).

Algorithm A ' s build orders resulted in significantly longer game times on average. This explains the increased minerals collected. Algorithm B had a relatively steady average match time amongst its build orders per generation. Since the scores across the generations for algorithm B seemed to remain the same on average, I graphed the win rate against a few of the scores.

The pyramid shape observed in Figure 8 indicates how a very high score doesn ' t produce more wins, but specific scores values indicate a high win rate of a build order. This looks like there is some other factor influencing wins.

5. Discussion

It is clear that both genetic algorithms produced more fit build orders, but algorithm A did not successfully create a significant amount of build orders that could win matches. Algorithm B ended up with several times the

amount of build orders that won at least a single match. From this experiment, we can move forward starting with algorithm B. Changes can be made to the fitness function and crossover method to increase the win rate further.

Looking closer at the build orders per generation of algorithm A, the fitness, build order length, match length, and resources collected of the build orders converges. This indicates that a single point crossover doesn ' t change build orders significantly after ten generation, given the set population and tournament selection. We believe this means a build order needs to be manipulated in smaller pieces rather than chunked into two halves.. For example, adding mutation, or multiple points for crossover. It seems even the change of one instruction, especially at the beginning of a build order, greatly affects the results of a match. A quick look at the winning build orders show they all sent a few marines very early on in a match, catching the enemy without defenses.

We speculate that by taking the second half of a build order and placing it first, done in algorithm B, allowed more initial build order instruction combinations to be tested which allowed more winning strategies to be found and improved. It appears algorithm B 's build orders had well timed attacks, which wasn ' t reflected in the chosen fitness function, nor was winning. Adding win rate to the fitness function, as well as scaling the other four scores would be another place to improve our genetic algorithms generation of winning build orders.

We plotted more scores against the win rate for algorithm B ' s build orders to show how most wins hover around specific build, unit, total minerals scores. This further shows there is other data unknown to our genetic algorithm influencing win rate. As mentioned before, it is most likely the timing of attack from our StarCraft bot. Information about how many marines are trained, how many are sent to attack at a time, and when they are sent are not known by the genetic algorithm. Algorithm B doesn ' t have any data converging as the generations pass, but it does appear that its build orders converged to very similar strategies. This means, as the other data indicates, our genetic algorithm needs to create more diverse build orders if we want to find many winning strategies.

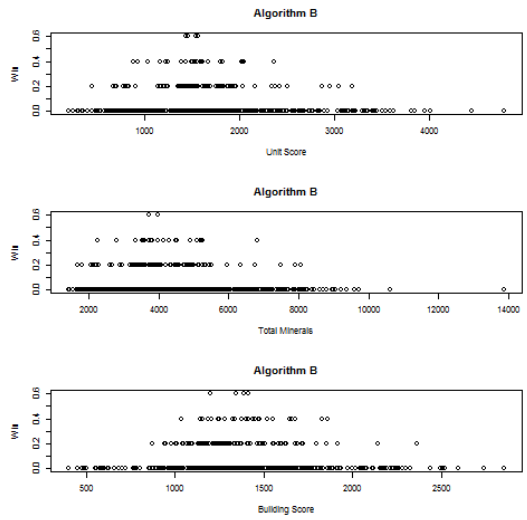

図 8 ユニット、リソー、建物スコアと手法 B の勝利率
Fig. 8 Unit Score, Minerals Score, and Building Score against Algorithm B's BO win rates

6. Conclusion

By comparing the algorithms we wrote, many places within our genetic algorithm arise as places to be tweaked or further studied. At first, it would seem writing a StarCraft bot and genetic algorithm to train it would be the most difficult portion of our project. On the contrary interpreting data it into a guide of what parameters to tweak proved most difficult. Seeing how far reaching a change in our algorithm, like changing the crossover function, can be exemplifies the complexity of learning how to win a match in StarCraft. It is intriguing to explore the idea of creating versatile solutions that solve an unknown set of problems. Discovering and improving the ability to do so, will lead to easier creation of even more complex AIs controlling NPCs in video games.

参考文献

[1] David Edward Goldberg, *Genetic Algorithms in Search, Optimization, and Machine Learning.* Boston: Addison-Wesley. (1989)

[2] Brood War API Team, *bwapi–An API for interacting with Starcraft: Broodwar (1.16.1)*, Online, https://code.google.com/p/bwapi/

[3] MasterOfChaos, *chaoslauncher–open source third-party launcher for StarCraft*, Online, http://www.teamliquid.net/forum/brood-war/65196-chaoslauncher-for-1161

[4] Computation Intelligence and Games, Retrieved July 25, 2017, from http://www.cig2017.com/competitions-cig-2017/

[5] J. Togelius, M. Preuss, N. Beume, S. Wessing, J. Hagelbck and G. N. Yannakakis, *Multiobjective exploration of the StarCraft map space*, Proceedings of the 2010 IEEE Conference on Computational Intelligence and Games, Dublin, pp. 265-272. (2010)