Evolving Gomoku Solver by Genetic Algorithm

Junru Wang[a], Lan Huang[b,*]
Computer Science Department
Yangtze University
Jingzhou, Hubei, China
[a]jkserge@163.com, [b]lanhuang@yangtzeu.edu.cn

Abstract— **Gomoku, also known as Gobang or five-in-a-row, is a popular two-player strategical board game. Given a squared 15×15 board, two players compete to first obtain an unbroken row of five pieces horizontally, vertically or diagonally. Classic methods for solving such games are based on game-tree theory, for example the minimax tree. These methods have a clear disadvantage: the depth of search becomes a bottleneck all the time. In this paper we propose a genetic algorithm for solving the Gomoku game. We investigated the general framework for applying genetic algorithm to strategical games and designed the fitness function from various game-related aspects. Empirical experimental results showed that the proposed genetic solver can search in greater depth than traditional game-tree-based solvers, resulting in better and more enjoyable solutions, and does so more efficiently.**

Keywords-artificial intelligence; gomoku; games; genetic algorithm; fitness function;

I. INTRODUCTION

Manufacturing artifacts that are as intelligent as humans has been an everlasting dream of the human race, and has motivated abundant advancements in both research and the general IT industry. Games are one of the most widely researched areas, especially in the strategical game genre. Researchers study methods to make computer game solvers more intelligent and more efficient. Game tree has become the mainstream method, for example the chess solver based on the minimax tree search has successfully defeated the best human chess players [1]. Essentially, these methods find the best next move by enumerating all possible moves and choosing the one that can bring the most favorable solution. The deeper a solver can search in a game tree, the more effective it is. Generally, three-layered structure is close to human mind and is sufficient to make simply inference and response: either to attack or defend in a strategical game. However, time and space complexity grows exponentially with search depth, which limits the real-time performance of the game-tree-based solvers, especially given the limited computing resource available to the general public.

Genetic algorithm (GA) is a heurist-based search method proposed by Holland [2] and widely applied to optimization problems. GA has been increasingly exploited for games in recent years. In contrast to tree-based algorithms, GA-based solvers do not need to enumerate all possibilities. Therefore, they can search deeper steps and do so within less time. In other words, they can be more intelligent and efficient than traditional solvers. In this paper, we explore the general framework for employing GA to games, and investigate the key components with the Gomoku game, which is a classic two-player strategical board game. The genetic solver desgined in this paper employs a general yet effective framework, searches the best move in the depth of seven levels and can play comparably with human players.

Next section introduces related work, then section 3 describes our GA-based Gomoku solver. Section 4 presents experimental setup and discusses the results. Section 5 concludes the paper.

II. RELATED WORK

Minimax search and its improvement the AlphaBeta search [3] are arguably the most classic and well-known game-tree-based methods for solving strategical computer games. Basically, in each turn these methods choose the most favorable move for the solver and the least favorable move for its competitor. AlphaBeta pruning further improves the search efficiency and has been widely applied to games.

Genetic algorithm performs search in a way similar to the natural selection process. Each possible solution is coded as an individual. Individuals evolve by crossing-over with other individuals and by mutating part of their genome. The evolving process is guided by a fitness function, which indicates the quality of an individual. Better individuals have greater chances to have their genomes selected and inherited by offspring.

As a result, GA does not require enumerating the entire search space, yet with a good fitness function it can function well. Actually, GA can be regarded as the optimization process of the fitness function. Applying GA to games has attracted increasing attention [4]. Han [5] applied GA to optimize the batting order of baseball teams. Cardamone, Loiacono and Lanzi [6] use GA to generate tracks in a racing game. Mitsuta and Schmitt [7] utilized GA to improve the performance of GNU-Chess and used the position-evaluation fitness function. Elyasaf, Hauptman and Sipper [8] used GA to help solving the FreeCell game and was able to reduce the number of search nodes and time cost, shorten the solution length, and outperform human players. Salge et al. [9] leveraged GA to enhance game fun and user experience. In this paper, we consider effectiveness, efficiency and fun factors in designing the GA-based Gomoku solver.

III. METHODOLOGY

The key to a successful GA game solver is to design an effective architecture for the problem at hand. Such

978-1-4799-6989-0/14/$31.00©2014 IEEE

architecture consists of five components: coding scheme for individuals, the original population size, crossover and mutation operations, and the fitness function. This section first describes the overall architecture and then explains each component in detail.

A. Architecture

Figure 1 shows the general architecture of the GA-Gomoku solver. Whenever the computer is to make the next move, it first generates a primitive population of individuals, where each individual corresponds to a candidate solution. These individuals are then evaluated by the fitness function. The salient ones are selected as the *seeds* and will be used to evolve the population, i.e. by crossover and mutation. The produced generation will again be evaluated and the evolution process continues until the best next move is found, i.e. when the population stabilizes.

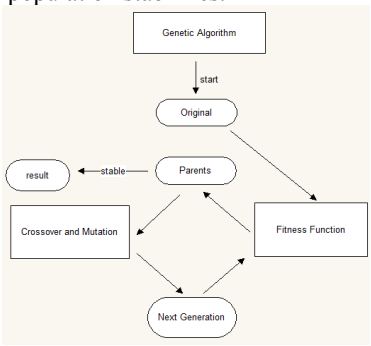


Figure 1. Computation process of the fitness function

B. Coding Scheme

The classic Gomoku board is 15×15, resulting in a total of 225 possible locations. In the GA framework, candidate solution is coded as a string. We use the coordinates of a piece and consider seven consecutive steps from the current layout. In other words, we search in the depth of seven levels to choose the best next move. This is much deeper than the traditional three-layer game tree structure.

Let ***A*** be the computer solver, and ***B*** the human player. Suppose human has just placed a piece at (7,5) and computer plays next, then the following two sequences correspond to the layout presented in Figure 1:

Sequence: ***B***(7,5) ***A***(9,4) ***B***(7,6) ***A***(6,4) ***B***(7,7) ***A***(8,4) ***B***(7,4) ***A***(7,8)
Representation: {(9,4), (7,6), (6,4), (7,7), (8,4), (7,4), (7,8)}

In this case, human has placed a piece at (7,5), and the GA-based solver generates the subsequent seven steps, starting with the computer placing a piece at (9,4). The player symbols ***A*** and ***B*** can be omitted for a simplified yet equally effective representation, which is adopted in the remaining sections.

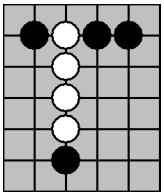


Figure 2. An example solution of Gomoku

C. Primitive Population

The diversity and the size of the primitive population is important for GA to find a good solution quickly. Usually individuals are generated randomly. Yet for Gomoku there are some restrictions. For example, it does not make sense to place a piece at the board boundary when all the other existing pieces concentrate in the center of the board.

Given the current layout, let (x_L, x_R) and (y_B, y_T) be the range along the horizontal and vertical dimensions of existing pieces on the board. We restrict the next seven pieces to be placed in the expanded boundary of (x_L-1, x_R+1) and (y_B-1, y_T+1). Pieces must be placed on vacant locations.

The primitive population is subsequently evaluated by the fitness function. Salient individuals that have high fitness values are kept whereas the others are discarded. The primitive population needs to be diverse and thus usually consists of thousands of individuals.

D. Crossover and Mutation

To evolve, selected individuals crossover with each other. Basically, better individuals are more likely to be selected and combined with other individuals, similar to the natural selection process. The roulette wheel selection algorithm is usually applied. The crossover point is chosen randomly. For example, the following two individuals crossover at their fourth genotype:

Parent 1: (7,6) (5,5) (5,4) (2,2) (6,5) (7,7) (4,4)
Parent 2: (5,7) (5,5) (2,4) (7,7) (3,5) (4,4) (3,3) ,

and generate the following combinations:

Child 1: (7,6) (5,5) (5,4) (7,7) (3,5) (4,4) (3,3)
Child 2: (5,7) (5,5) (2,4) (2,2) (6,5) (7,7) (4,4) .

Mutation point is also chosen randomly and the total number of mutated individuals account for no more than one percent of the population. It is worth noting that both crossover and mutation can produce invalid individuals: duplicate genotypes in a single individual. This means that a piece is placed at a non-vacant location. Although it is possible to avoid this by double checking the generated candidates, efficiency will be compensated. Therefore, we leave it to the evaluation step where such candidates will eventually be discarded by the fitness function.

E. Fitness Function

Fitness function assesses the quality of a candidate solution, and is the most important component of a GA-based program. Distinguishing from other search problem, strategical games are more complex because they involve multiple players. We design the Gomoku fitness function from five aspects, as described below.

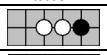
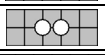
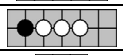
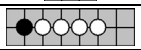
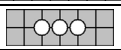
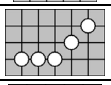
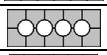
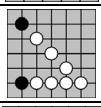
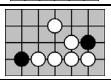
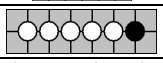
Firstly, we categorize the possible layouts into 12 types and assign values to each type according to the likelihood of winning. Specifically, all patterns are first sorted in ascent order of their likelihood, and their values are computed using the following formula:

$$v(p_n) = 2 \times v(p_{n-1}) + 1, \quad (1)$$

where p_n and p_{n-1} are consecutive patterns and function $v(p)$ computes pattern values. Table I lists the twelve patterns and their values. Patterns 9, 10 and 11 have the same value, simply because they are all one step away from victory: the

computer is guaranteed to win the game regardless of the competitor's next move. Layouts that cannot be classified into these twelve patterns are assigned zero values.

TABLE I. LAYOUT PATTERNS AND VALUES

ID	Pattern	Value
1		1
2		3
3		7
4		15
5		31
6		63
7		127
8		255
9		511
10		511
11		511
12		1023

Secondly, besides the attack values computed above, we also compute the defense values, by assuming that the competitor takes the next move instead. For example, in the layout showed in Figure 3, assuming the computer is taking black and it is assessing the three locations marked with numbers. For location 1, the attack value is simply the resulting pattern's value described above, which is 63 (pattern 6). Its defense value is the pattern value that given its competitor takes this place instead, which is 511 (pattern 9). Therefore, the total value of location 1 equals the sum of its attack and defense values, i.e. 574. Similarly, the attack values for locations 2 and 3 are 0 and 63 respectively, and the defense values are 511 and 0, resulting the total values of 511 and 63. Clearly, location 1 is the most preferable. This is also why subsequent pattern's value is defined based on two times of the previous pattern's value in formula (1).

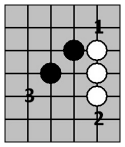


Figure 3. Example of attack and defend values

Given the above definition, we can define the primitive fitness function as

$$f(c)=\sum_{i=1}^{7}(atk(p_i)+def(p_i)), \qquad (2)$$

where i indexes into the seven steps considered in each individual, and $atk(p)$ and $def(p)$ are the attack and defense values of pattern p respectively.

The third aspect considered is to favor early victories. When five in a row occurs, subsequent steps become redundant. For example, given the situation in Figure 4, for the player using the white pieces, the most favorable two locations are marked. For location 1, it directly settles the game, yet its value is 1023 for attacking and 0 for defensing. In contrast, location 2 scores 511 for attacking and 1023 for defensing, which sums to 1534. As a result, location 2 will be undesirably selected instead. Therefore, in order to acknowledge the value of early victories, we directly assign the maximum value to the redundant steps. As a result, the sooner the computer achieves victory, the higher the fitness value will be.

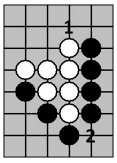


Figure 4. Example of an early victory

The fourth aspect deals with the case of close-victory. For example, when patterns 9, 10 or 11 occur, the player with the white pieces is guaranteed to win. Therefore, we also assign the maximum value to subsequent steps, so as to ensure that such close-victory cases will be selected by the GA-solver.

Last but not least, we consider the overall tendency of an individual: whether it is more conducive to the computer or the competitor. Formula (2) does not differentiate between the two players; in other words, it might also assign high value to a situation that is actually in favor of the competitor. Therefore, we quantify the overall tendency using the following formula

$$t(c)=\sum_{i=1}^{7}l(i)\times[atk(p_i)+def(p_i)], \qquad (3)$$

where function $l(i)$ equals to 1 if the ith step is taken by the computer and −1 otherwise. A negative value means that the current situation is hardly conducive to the computer solver. Therefore, the solver ought to take its competitor's next move, that is, place its next piece at the location suggested by the second genotype rather than the first genotype. This is particularly helpful to improve the intelligence of the solver and thus enhance game fun.

IV. EXPERIMENTS AND ANALYSIS

We implemented the proposed GA-based Gomoku solver in Delphi and made it available here[1]. In this section we evaluate it by comparing its solution and search costs to the classic game-tree-based solver. All experiments were performed the on the same computer.

The primitive population size is set to 2000 and the maximum population size is 3500. In each iteration, the top

[1] http://sourceforge.net/projects/gagomuku/

200 individuals with the greatest fitness values are kept to produce the next generation. One percent of the population involves a mutated genotype. The best individual in each generation (i.e. iteration) is kept, and the genetic algorithm converges if the first genotype of these best individuals stabilizes for five consecutive generations, i.e. when the best next move stabilizes.

A. *Converge Speed*

Gomoku is a live strategical game and short response time is crucial. On average, it takes 0.6 seconds for the solver to find the best solution. Actually, we found that the GA-based solver converges quickly. Table II shows the fitness values of the best individual in each generation for five independent tests. Clearly, the best solution keeps evolving, as indicated by increasing fitness values. Except for one case, all the tests converge within seven runs, and results in Table II show that they converge quickly.

TABLE II. BEST FITNESS VALUES ACHIEVED IN EACH GENERATION

Generations	1	2	3	4	5	6	7	8	9
Test 1	20505	20509	20509	20511	20517	20517	20517	20520	20520
Test 2	30667	30698	30698	30698	30698	30698	-	-	-
Test 3	20496	20618	20618	20619	20619	20619	-	-	-
Test 4	20636	20636	20636	20644	20644	-	-	-	-
Test 5	20741	20741	20741	20744	20744	20759	20759	-	-

These findings suggest that the proposed GA-based Gomoku solver is able to find reasonably good solutions. Actually, the solver beats human player in most cases. Furthermore, with genetic algorithm, we no longer need to exhaustively search in the immerse space of possible solutions, thereby significantly reduce time and space complexity. Our empirical experimental results demonstrate the advantages of using genetic algorithm to solve game problems.

B. *Number of Search Nodes*

We also compared the number of search nodes explored by different solutions, as shown in Figure 5.

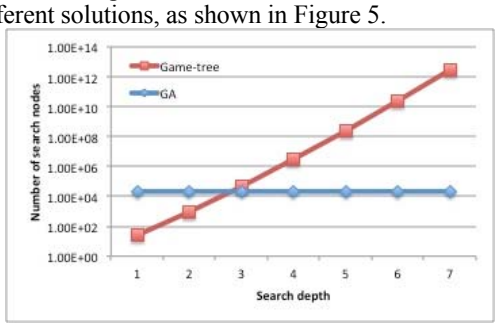


Figure 5. Comparing the search costs of the game-tree and genetic solvers

On average, the GA-solver takes 5.6 iterations to converge. The average number of search nodes is the maximum population size 3500 times the average number of iterations 5.6 and plus the primitive population size 2000, which equals to 21,6000 nodes. Search depth, i.e. the length of the encoded individuals, has no impact on the complexity. The criterion for convergence—the first genotype stabilizes for five consecutive runs—clearly impacts the complexity. Results in Table II show tiny differences between iterations, and this is consistent with our empirical observation that the first couple iterations already can find the best next move.

In contrast, the number of search nodes explored by the game-tree solver increases steadily with search depth. Therefore, game-tree-based solvers are limited by this exponential increase rate.

V. CONCLUSION

In conclusion, in this paper we investigated methods for employing genetic algorithm to solve strategical games. We highlighted several important aspects in designing the genetic fitness function: the value of layout patterns, differentiating between attacking and defending values, adjustments for early victories and close victories, and the overall tendency of the solution. Empirical experimental results have demonstrated the advantages of the proposed genetic solver over the traditional game-tree-based solver. The genetic solver not only can search in greater depth, resulting in better solutions, more enjoyable game experience and comparable performance with human players, but also is more efficient. The genetic framework and the fitness function can easily be extended to other strategical games. Besides, we also make the solver publicly available at http://sourceforge.net/projects/gagomuku/.

REFERENCES

[1] F. H. Hsu, Behind Deep Blue: Building the Computer that Defeated the World Chess Champion. Princeton University Press, 2002.

[2] J. H. Holland, Adaptation in Natural and Artificial Systems. Arbor:University of Michigan, 1975.

[3] T. Ibaraki, Y. Katoh, "Searching minimax game trees under memory space constraint," Annals of Mathematics and Artificial Intelligence, 1(1-4), pp. 141–153, 1990.

[4] Z. X. Ma, Y. Li, Y. Ch. Lu, "Using genetic algorithm to solve game problem," Henan Sciences, 25(2), pp. 273–276, 2007. [in Chinese]

[5] S. Han, "Batting order optimization by genetic algorithm," In Proceedings of GECCO'12 Companion, pp. 599–602, 2012.

[6] L. Cardamone, D. Loiacono, P. L. Lanzi, "Interactive evolution for the procedural generation of tracks in a high-end racing game," In Proceedings of GECCO'11, pp. 395–402, 2011.

[7] T. Mitsuta, L. M. Schmitt, "Optimizing the performance of GNU-Chess with a genetic algorithm". In Proceedings of HC'10, pp. 124–131, 2010.

[8] A. Elyasaf, A. Hauptman, M. Sipper, "GA-FreeCell: Evolving solvers for the game of FreeCell," In Proceedings of GECCO'11, pp. 1931–1938, 2011.

[9] C. Salge, C. Lipski, T. Mahlmann, B. Mathiak. "Using genetically optimized artificial intelligence to improve gameplaying fun for strategical games," In Proceedings of Sandbox Symposium 2008, pp. 7–14, 2008.