Genetic Programming Produces Strategies for Agents in a Dynamic Environment

Robert E. Keller Walter A. Kosters Martijn van der Vaart
Martijn D.J. Witsenburg

LIACS — Leiden Institute of Advanced Computer Science
Universiteit Leiden
P.O. Box 9512, 2300 RA Leiden, The Netherlands
{rkeller, kosters, mvdvaart, mwitsenb}@liacs.nl

Abstract

We introduce an agent-based approach to game playing where every agent is an element of a game and is driven by a control algorithm that is being adapted by Genetic Programming, without using an explicit fitness function. A software system representing this approach is implemented for two simple variants of chess, while the approach is applicable to all agent-related problem domains, such as economics. In particular, we focus on individual rule learning and behavior adaptation to a given environment such that communication and cooperation emerge among agents. Empirical results are given.

1 Introduction

Game playing is of major interest to approaching real-world decision problems, since games capture essential properties of problem environments, such as change, limited knowledge and resources, competition and cooperation. Thus, strategy evolution in artificial dynamic worlds with incomplete knowledge and many decision makers can be applied to real worlds like economics, management, defense, and sociology. Games may model such worlds when real entities are simulated by agents, rules, and game resources.

In the context of the present approach, an agent makes decisions under control of an individual control algorithm which is adapted by *Genetic Programming* [1]. Genetic Programming (GP) is a heuristic probabilistic optimization process that creates and improves representations of computer programs by use of algorithmic metaphors of natural evolution.

An implementation of the approach presented here is developed as a self-organizing software system in which learning and cooperating yet selfish agents emerge. As the game of chess captures numerous properties of real environments, simple variants of this game are crude real-world models employed in the system.

Opposed to deterministic chess programs, the system described here does not feature a purely human-designed algorithm. Also, the system does not represent a probabilistic algorithm that creates chess players which have complete chess knowledge and which centrally control dumb pieces as it is described in the interesting work [2]. Instead, the system represents a probabilistic algorithm that creates initially dumb agents (representing pieces such as a king) which may learn the game rules from scratch and act autonomously. The actions of an agent are directed by a set of individual *control algorithms* which are being adapted to the permanently changing environment by Genetic Programming. It is a hard side constraint that all *learning agents* have incomplete domain knowledge, i.e., neither do they know game rules nor the objective of the game. In particular, the system design implies that there is no entity which represents a central chess player.

The project background is an interest in modeling, explaining and influencing multi-objective real worlds, while the models are not human-designed. Instead, such a model comes into existence as a set of co-evolving agents which permanently change their behavior in response to the changing behavior of other learning agents in the set. Phenomena like competition, cooperation and self-maintenance of agents, resulting from pleasure-gaining and pain-avoiding behavior, are relevant. In particular, the present ambition is not to engineer a specialized chess-playing algorithm but to devise a system which produces agents that adapt to changing rules and situations. While given fixed rules model constant environmental rules such as physical laws, dynamic rules come into existence as results of the interaction of agents. This degree of freedom allows for modeling real-world domains featuring autonomous decision makers, such as economical and social networks. Such emergent agent behavior which fosters the self-maintenance of a learning agent in a dynamic environment may then be called "intelligent".

As a system design principle, we adhere to *minimalism*: in case of a choice to be made between design options, we choose the simplest one since we are primarily interested in phenomena which have not been designed to occur. In particular, contrary to most Genetic Programming instances, no explicit quality measure for evolved algorithms, i.e., a *fitness function*, is used.

First, we shall introduce simple chess variants and explain the working of the system in some detail. In particular, we discuss how an evolved control algorithm represents a strategy which decides the next move of a learning agent. We then suggest several additional techniques, describe empirical results and give conclusions and directions of further research, while more details can be found in [3].

2 Poor Man's Chess and Pseudo Chess

We assume basic knowledge of *Standard chess* on the side of the reader. We introduce other chess variants by explicitly mentioning modifications of this standard.

A simple chess variant shall be called *Poor man's chess*, and in this variant two agent teams, called *Black* and *White*, take turns on a board with $n \times n$ squares ($n > 1$). Both teams have one king each and $n - 1$ pawns, and a king can move to any adjacent square whereas a pawn is allowed to move one or two squares in any direction. Pawns are not allowed to move through an occupied square, and a pawn may move onto an occupied square if and only if the occupant is of the opposite team, thus taking the occupant. A game ends when a king is taken, and the team whose king has been taken is considered to have lost, while the other team is considered to have won. If, after a game-specific fixed number of moves, the game has not ended this way, it ends with a draw.

Another game variant shall be called *Pseudo chess*; it features all chess pieces and most of their standard-chess moves. A king may stay in check, the 50 moves rule does not apply, a promotion of a pawn results in a random-selected piece with no initial knowledge, and castling and taking *en passant* are illegal.

3 System Setup

Every learning agent in a game represents an agent-specific population of *popsize* control algorithms which are called *individuals*, and two preliminary definitions of learning shall be given. *Learning of an agent* means an adaptation of its population such that the average survival time per game of this agent rises over several games. *Learning of a team* means that the quotient of the number of games won by the team and the number of games lost by the team increases over time.

For each *experiment* presented below, i.e., a non-empty series of games, *popsize* typically equals 40. Every individual is represented as a string that defines a strategy which, if activated, suggests the next move of the agent. This situation is depicted in Figure 1, and the semantics of the string representation is explained in detail in Section 4.

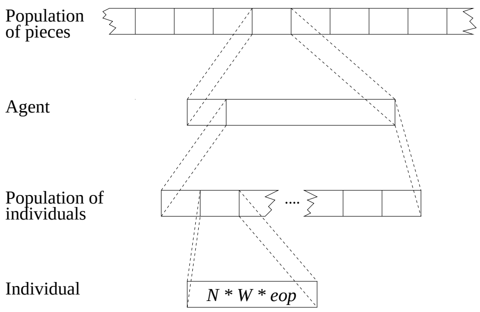

Figure 1: A game features agents, a learning agent is represented as a population of individuals, and an individual is a special-purpose string.

A white agent is always a learning agent, while a black agent may be a learning agent or is under the control of a fixed human-designed strategy as described in Section 5. The individuals in a population are randomly initialized: each individual initially represents a syntactically correct random control algorithm. Such an algorithm may suggest an illegal move, since there is no initial knowledge of the rules of the game. Thus, it is significantly harder for an agent to learn the game rules than it usually is for a human to whom another human explains the rules.

If a white agent has to make a move, the system i) first random-selects a white agent and then ii) random-selects an individual of this agent. If this individual suggests a legal move, the agent executes this move, while, otherwise, the system continues with i), and this procedure is repeated until a legal move is suggested. In order to decide the legality of a suggested move, the system has complete knowledge of all game rules — but no knowledge about playing a game elegantly. This scenario reflects the real-world situation where knowing the rules of an environment is not identical to knowing how to operate smoothly within this environment. The system does not give explicit feedback to a learning agent with respect to the quality of a move: it only feeds back binary information on the legality of a suggested move.

As soon as an individual has suggested a legal move, the corresponding agent $\mathcal{A}$ executes the move. Then $\mathcal{A}$ *communicates* by sending a copy of the individual that suggested the move to $\mathcal{A}$'s direct neighbors of the same color, if any. *Direct neighbors* are agents that are within sensor reach of $\mathcal{A}$ immediately *before* or *after* the move. Such a neighbor replaces its own individuals, each with some probability (typically 25%), by the communicated individual. This process is referred to as *communication*.

Finally, each individual of each learning agent (in particular individuals changed during communication) may or may not be changed in a random way. Such a change and the process effecting the change are called a *mutation*; the probability that an individual is to be changed is called the *mutation rate*, and it has the order of magnitude of 10%. If an individual of an agent has suggested an illegal move n times in a row, n being an experiment-specific parameter, then the individual definitely undergoes mutation since it seems to be useless. Note that there is always a degree of freedom for changing an individual, so that, after a change, the individual indeed differs from its original.

After a game has ended, all agents, taken as well as not taken during the game, are set back into their initial positions on the board, and the system is ready for a new game. For the time being, we decided against reinitializing parts of the individual-populations of agents in order to have learning from previous unwanted behavior occur. Thus, the population of an agent represents the continuous *memory* this agent has of its past.

As implied by the above description, there is no explicit fitness function which would feed the quality of an individual back to the system. Instead, if a suggested move is legal, this situation is implicitly rewarded in the sense that the suggesting individual is copied to neighboring agents. Illegal moves are implicitly punished by not proliferating the suggesting individual. Thus there is no need for a mechanism that smoothly handles all situations in a meaningful way. The design of such a mechanism seems impossible in the context of a complex dynamic environment.

Some explicit information — in particular, parameter values — must however be provided by the system user. In order to improve this disadvantageous situation, the mutation rates employed by the system can themselves be adapted by use of mutation. Thus, there may occur *co-evolution* of individuals and mutation rates, like there is co-evolution occurring among agents.

4 Representation

An individual represents a move-suggesting strategy that is coded as a character string. We now describe a formal language containing all such strings. The terminal symbols of this language are:

- A *direction symbol* indicating one of the eight possible directions of a move: N, Ne, E, Se, S, Sw, W and Nw.
- *Sensor input* to a scanning agent: *nil* (no neighbor square found in current direction), *empty* (empty neighbor square found in current direction), $+t+c$, $+t-c$, $-t-c$ and $-t+c$ (occupied neighbor square found in current direction, where $+t$ denotes an agent of the same type as the scanning agent, $-t$ an agent of a different type, $+c$ an agent of the same color as the scanning agent, and $-c$ an agent of a different color).
- Reserved grouping symbols: { and } (embracing Boolean conditions), (and) (embracing if- and else-part within a Boolean condition), : (separator within a Boolean condition), $*$ (optical indicator of an end of a move) and *eop* (end of an individual).

An individual consists of one or more move suggestions, followed by *eop*. A move suggestion either results from a direction symbol or from the evaluation of a Boolean condition followed by two direction symbols: the first for the if-part and the second for the else-part. The condition itself consists of a direction symbol followed by a sensory input. Nested constructions are possible. We give some examples of "strategies":

- $\{N : nil\}(S*)(W*) * eop$ — If there is no neighboring square to the north, then go one square to the south, else go one square to the west.
- $Se * \{E : +t-c\}(E*)(Nw*) * eop$ — Go one square to the southeast. If there is an agent of the same type but of the opposite color on the neighboring square to the east, then move there (thus taking the agent), else go back to the original square.

Note that the second strategy suggests a knight move in case an opposing agent can be taken this way.

The described language elements are potential parts of all individuals, while there is a language extension called *Expanded GP*. This extension features elements representing sensor inputs indicating an opposing king or pawn. It also features the direction symbols Di and oDi that remember the direction that occurs in the Boolean condition and the according opposite direction [3].

We focus on mutation for introducing variation into the population of a learning agent, and a mutation replaces a move suggestion with a random-created different move suggestion.

5 Additional Mechanisms and Strategies

In order to experiment with different instances of the described system, some additional mechanisms are introduced. We also present the human-designed mechanisms controlling the behavior of the opponents of the learning agents.

Timetable is a method which selects the learning agent that is expected to execute the next move of its team. This method may replace the random method described in Section 3. A selected agent itself computes a time slot for its next move suggestion, e.g., by adding a fixed number t of time units to the current time slot. The agents can now be scheduled without conflicts in a suitable order. One can also use agent communication to have this order created, and this approach is referred to as the *extended* Timetable method; details can be found in [3].

During a game, opponents of the learning agents may act under the control of one of the following two fixed central random strategies:

- i) Select a piece at random. ii) Generate a random control algorithm in the representation given in Section 4. iii) If this algorithm computes an illegal move, then go to ii), else move the selected piece.
- i) Select a piece at random. ii) Select, at random, an available direction and distance. iii) If this does not produce a legal move, go to ii).

Besides these two random strategies, we designed, for the case of Poor man's chess, a fixed central strategy called *Clean Sweep*. Here, all black pieces move to south as often as possible, then they move north as often as possible, and so on. For Pseudo chess, a fixed central random strategy is used which employs the *Magnet principle*, i.e., black pieces are attracted toward white agents.

6 Results

We present the results of experiments with system instances as described above.

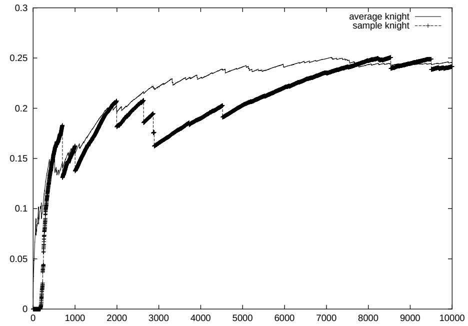

Figure 2: Fraction of legal moves suggested by a knight agent.

In Figure 2, results for Pseudo chess are summarized. This visualization shows i) the fraction of legal moves out of all suggested moves for the same sample knight agent for one series of games ("lower" graph) and ii) the average of these fractions over ten series of games. It can be seen that the agent learns to suggest an increasing number of legal knight moves. The oscillations of the displayed graph may be due to situations with only some pieces left and few possibilities to learn.

In Table 1, we show results for Poor man's chess for the white team, whereas the black team employs the second random strategy. Every table row represents exactly one initialization of the white team and shows the number of wins for both teams in the series of games under consideration. When percentages do not add up to 100%, this means the remaining games ended in a draw. The 50% – 50% behavior shows that the learning agents often behave similar to the black pieces, i.e., randomly, while the Timetable method significantly improves the performance.

Number of games	Black wins	White wins	Communication	Timetable	Expanded GP
100,000	50.8%	49.2%	before	no	no
100,000	52.2%	47.8%	before	no	yes
100,000	51.8%	48.2%	after	no	no
100,000	50.7%	49.3%	after	no	yes
100,000	50.9%	49.1%	before	no	no
100,000	52.1%	47.9%	before	no	yes
100,000	50.3%	49.7%	after	no	no
100,000	51.9%	48.1%	after	no	yes
100,000	38.7%	61.3%	before	yes	no
100,000	22.7%	77.3%	before	yes	yes

Table 1: Results of experiments with Poor man's chess.

Number of games	Black wins	White wins	Random strategy	Communication	Timetable	Co-evolution
100,000	69.1%	30.9%	second	after	no	no
100,000	68.2%	31.8%	second	after	no	no
100,000	59.6%	40.4%	first	after	yes	no
100,000	59.2%	40.8%	first	after	yes	no
100,000	44.2%	55.6%	first	after	yes	yes
100,000	45.7%	54.3%	first	after	yes	yes
100,000	52.0%	47.9%	second	after	yes	yes
100,000	52.6%	47.3%	second	after	yes	yes
100,000	62.7%	37.2%	first	before	no	no
100,000	60.4%	39.6%	first	before	no	no
100,000	57.6%	42.4%	first	before	yes	no
100,000	58.8%	41.1%	first	before	yes	no
100,000	51.0%	49.0%	first	before	yes	yes
100,000	68.7%	31.0%	first	before	yes	yes
100,000	58.4%	41.4%	second	before	yes	yes
100,000	71.9%	27.8%	second	before	yes	yes

Table 2: Results of experiments with Pseudo chess.

Table 2 shows results in the case of Pseudo chess, and experiments are performed twice with the same settings but with different seed values for the pseudo-random number generator. In most situations, the random strategies outperform the learning agents. This might be explained by the fact that different learning agents have different probabilities of suggesting a legal move, while all learning agents have the same probability of being selected. Wins for the learning agents occur when the Timetable method is used and communication takes place afterwards. This result again shows the importance of the Timetable method which allows for a different selection mechanism. Clearly, inter-agent communication is critical because there is potential in the coordination of agents. Further improvement can be expected from explicit rewards in case of a win.

Number of games	Black wins	White wins	Communication	Timetable	Expanded GP
5,000	62.9%	36.8%	before	no	yes
5,000	63.2%	36.7%	before	no	yes
5,000	40.8%	58.9%	before	yes	yes
5,000	40.4%	58.9%	before	yes	yes
5,000	24.6%	74.2%	before	extended	yes
5,000	24.6%	74.0%	before	extended	yes

Table 3: Results of experiments with Poor man's chess versus Clean Sweep.

We finally give some results for Poor man's chess where the black pieces are controlled by the Clean Sweep method described in Section 5 (see Table 3). Note that the extended Timetable method produces the best performance.

7 Conclusions and further research

We introduced and examined a system of learning agents in a dynamic environment with Genetic Programming driving the learning process. A major result is that agent communication is critical for a learning rate superior to random learning.

Further research should explore improvements of agent communication, such as the introduction of new agents that select moving agents. By extending the presented formal language, for instance by using goto-statements or loops, more complex agent-motion patterns, such as rook- or bishop-like moves, may be produced. The system could then be applied to simple well-known chess end games in order to compare the produced solutions to known ones. Also, it is possible to incorporate the exchange of information between single individuals in order to introduce variation to the individual populations (crossover). Learning agents that often suggest legal moves could then distribute their knowledge to their team.

Finally, a real-world problem must be modeled as an environment in which learning agents of our system may develop strategies to approach the problem.

References

[1] W. Banzhaf, P. Nordin, R.E. Keller, and F.D. Francone. *Genetic Programming, An Introduction.* Morgan Kaufmann Publishers, Inc., 1998.

[2] R. Groß, K. Albrecht, W. Kantschik, and W. Banzhaf. Evolving chess playing programs. In *Proceedings of the Genetic and Evolutionary Computation Conference, GECCO 2002*, pages 740–747, 2002.

[3] M. van der Vaart. Distributed gameplaying using Genetic Programming. Master's thesis, LIACS, Universiteit Leiden, The Netherlands, 2002.