Genetically Optimizing the Speed of Programs Evolved to Play Tetris

Advances in Genetic Programming: Volume 2
edited by P.J. Angeline and K. Kinnear

Eric V. Siegel
Dept. of Computer Science
Columbia University
New York, NY 10027
evs@cs.columbia.edu

Alexander D. Chaffee
EarthWeb LLC
3 Park Avenue, 38th Floor
New York, NY 10016
alex@earthweb.com

Many new domains for genetic programming require evolved programs to be executed for longer amounts of time. For these applications it is likely that some test cases optimally require more computation cycles than others. Therefore, programs must dynamically allocate cycles among test cases in order to use computation time efficiently. To elicit the strategic allocation of computation time, we impose an *aggregate computation time ceiling* that applies over a series of fitness cases. This exerts time pressure on evolved programs, with the effect that resulting programs dynamically allocate computation time, opportunistically spending less time per test case when possible, with minimal damage to domain performance. This technique is in principle extensible to resources other than computation time such as memory or fuel. We introduce the game *Tetris* as a test problem for this technique.

14.1 Introduction

Many new domains for genetic programming require evolved programs to be executed for longer amounts of time. For example, it is beneficial to give evolved programs direct access to low-level data arrays, as in some approaches to signal processing [Teller 1996], and protein segment classification [Handley 1995; Koza and Andre 1996]. This type of system automatically performs more problem-specific engineering than a system that accesses highly preprocessed data. However, evolved programs may require more time to execute, since they are solving a harder task.

One way to control the execution time of evolved programs is to impose an absolute time limit. However, this is too constraining if some test cases require more processing time than others. To use computation time efficiently, evolved programs must take extra time when it is necessary to perform well, but also spend less time whenever possible.

In this chapter, we introduce a method that gives evolved programs the incentive to strategically allocate computation time among fitness cases. Specifically, with an *aggregate computation time ceiling* imposed over a series of fitness cases, evolved programs dynamically choose when to stop processing each fitness case. We present experiments that show that programs evolved using this form of fitness take less time per test case on average, with minimal damage to domain performance. We also discuss the implications of such a time constraint, as well as its differences from other approaches to *multi-objective problems*. The dynamic use of resources other than computation time, e.g. memory or fuel, may also result from placing an aggregate limit over a series of fitness cases.

The following section surveys related work in both optimizing the execution time of evolved programs and evolution over Turing-complete representations. Next we introduce the game Tetris as a test problem. This is followed by a description of the aggregate computation time ceiling, and its application to Tetris in particular. We then present experimental results, discuss other current efforts with Tetris, and end with conclusions and future work.

14.2 Background

14.2.1 Optimizing The Computation Time of Evolved Programs

We can divide approaches to optimizing the computation time of evolved programs into two categories: post-hoc, and genetic. Post-hoc methods are based on traditional techniques for optimizing software, and operate on a program after it has emerged from the genetic algorithm. For example, one can remove redundant or unused code [Koza 1992; Siegel 1994], or compile evolved programs into more efficient representations [Langdon 1994; Harris 1996]. However, most approaches optimize computation time genetically, including the approach taken in this chapter. This is achieved by incorporating time pressure into the fitness measure. This second approach may be preferable because the genetic search process examines the problem domain and may discover short cuts and idiosyncrasies missed by post-hoc techniques.

Methods that increase the parsimony of evolved programs may lead to a decrease in the computation time of evolved programs. For example, Ryan incorporates program size into the fitness measure when evolving sorting networks [Ryan 1994]. This introduces pressure for evolved sorting networks to have less comparison operations. Since a sorting network with fewer comparisons takes less time, its size directly influences execution time. Increasing parsimony may also increase the generalization potential and the comprehensibility of evolved programs.

However, program size is not always directly related to execution time. For example,

since some implementations of **IF-THEN-ELSE**, **AND**, and **OR** do not always require the evaluation of all their arguments [Koza 1992], executing a large evolved program may only entail the execution of a small number of instructions. As another example, if the evolved program tree is evaluated multiple times, the execution time depends on the number of evaluations, which is not always predictable. Therefore, increasing parsimony is not a reliable method for optimizing the execution time of evolved programs.

The Tierra system [Ray 1991] demonstrates the genetic optimization of evolved program speed. A self-reproducing program with 80 instructions was written by hand, and in one run of the system, it evolved into a self-reproducing program of only 22 instructions. Because evolving programs are executed in parallel and compete for resources, there is pressure to increase program speed. However, in contrast to standard genetic algorithms, there is no manually-imposed fitness measure in Tierra; self-preservation is the only goal.

The *coroutine execution model* [Maxwell 1994] also imposes pressure for evolved programs to have lower computation times. In this steady state model [Syswerda 1989], both evolutionary operations and the fitness measurements of the population are executed in parallel. Since evolutionary operations are guided by partial fitness measurements, there is an advantage for those individuals that accrue partial fitness faster. In addition to encouraging faster evolved programs, this model avoids the need to impose an upper bound on the execution time of evolved individuals. Further, invalid individuals are likely to be replaced before the time is spent to complete their fitness measurements. However, this model is only applicable to problems that support comparisons between incomplete fitness measurements.

In the Tierra system and the coroutine execution model, the time pressure imposed on an evolved program is not constant and is not explicitly controlled, since it is relative to the performance of the rest of the population. This leaves open the issue of whether or not evolved programs can allocate their computation time more efficiently if time pressure is explicitly imposed.

14.2.2 Indexed Memory and Evolving Algorithms

The representation for evolving algorithms used in this chapter is the parse tree of canonical genetic programming with the addition of indexed memory [Teller 1994a]. This is accomplished by introducing operations, READ and WRITE, that have random access to a "scratch pad" array of integers. With indexed memory available, and repeated parse tree evaluations, this representation is Turing-complete [Teller 1994d], and was the first such representation used for genetic programming. The viability of this representation and its subsequent variations has been demonstrated for signal processing problems [Teller 1996].

Genetic programming with indexed memory was first tested on the Tartarus problem

[Teller 1994a]. For this problem, programs are evolved to control the movements of an agent that resides on a six by six grid, and tries to push randomly positioned blocks to the walls and corners. The agent has eight sensors describing its neighborhood, but has no global information such as its position on the grid. Therefore, the availability of memory is crucial for this problem since the agent must be able to keep track of its position, if not also remember where it has seen blocks.

Because of the halting problem, an upper bound on execution time must be imposed when measuring the fitness of a program described in a Turing-complete representation. This can be achieved by imposing on each fitness case an absolute time limit or a time limit that is relative to the performance of the rest of the population [Teller 1994b]. Despite this upper limit, is it possible for evolved programs to halt spontaneously, before they reach the limit? Halting spontaneously would be necessary for evolved programs to spend varying amounts of time on test cases, allocating computation time dynamically.

Optimizing the computation time of evolved programs may be particularly beneficial when evolving Turing-complete algorithms. The advantage of a Turing-complete language is that any conceivable algorithm can be represented. Therefore, in principle, evolved programs can successfully use low-level operations that directly access raw data. In this case, it is no longer necessary to manually create high-level, domain-dependent operations that require the data to be preprocessed. This favors a *weak* approach to gaining task-specific knowledge over a strong approach [Angeline 1994]. Since this weak approach relies on evolved programs to perform more of the work than programs that use high-level operations, resulting programs may require more exeuction cycles.

14.3 Test Problem: Tetris

The experiments presented in this chapter are tested on the game Tetris[1]. The object of Tetris is to pack geometric shapes (*pieces*) together on a grid (the *game board*). One of seven possible pieces randomly appears at the top of the game board and "drops" to a resting position. This occurs for a random series of pieces, and the player selects a horizontal position and rotational orientation for each piece before it drops so that the board becomes filled as densely as possible. Figure 14.1 shows three sequential states of the Tetris game board, and Figure 14.2 shows the seven possible piece types. The game board is 10 by 24 blocks in size, with white boxes indicating the boundaries.

If and when a row on the game board becomes completely filled, it disappears, and

[1]Tetris, invented by Alexey Pazhitnov, is a trademark of AcademySoft and ELORG. For an example implementation see *http://www.stinky.com/tetris/* or *http://www.earthweb.com/java/Netris/*.

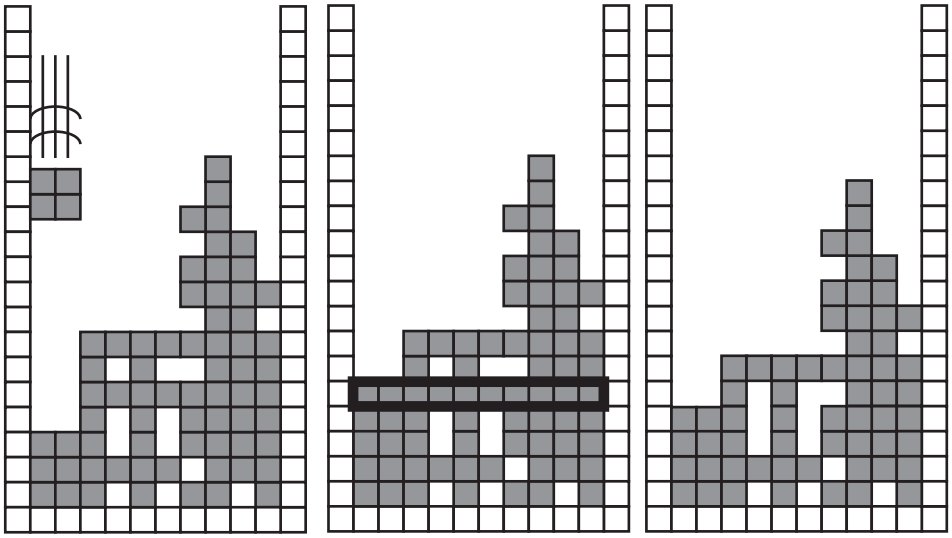

Figure 14.1
An evolved algorithm clears a line playing Tetris.

every block above it shifts down one row. This is called "clearing a line." For example, in Figure 14.1, the square piece drops from the top of the board, as shown in the first board state. It then lands, completely filling the fifth row from the bottom, as shown in the second board state. This row is cleared, and the result is shown in the third board state.

A game of Tetris ends when a piece is dropped to a position such that any block in the top four rows of the game board is filled, which can only happen when there is not enough room below the piece for it to fit within the bottom 20 rows. The game score is the number of blocks filled during the game, minus the number of "holes" (an unfilled block with at least one filled block above it) at the end of the game.

Note that this is a variation of the video game that many readers will have played. The piece does not drop incrementally, but instead drops all at once, *after* the player has determined the position from which it is to drop. Also, the player has the same amount of time to select a drop position for each piece, which contrasts with standard Tetris in which there is more time pressure as the board becomes filled.

14.3.1 Applying Genetic Programming to Tetris

The details for applying genetic programming to Tetris are shown in Table 14.1. The function set is composed primarily of standard genetic programming functions [Koza

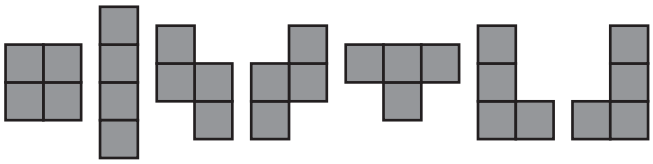

Figure 14.2
The seven possible piece types for Tetris.

Table 14.1
Evolving an algorithm to play the game Tetris.

Objective:	Evolve an algorithm to play Tetris.
Terminal set:	PIECE-TYPE, PIECE-PHASE, NEW-PIECE, SENSOR, FLOOR, CEILING, RIGHT-WALL, LEFT-WALL, and RANDOM-CONST.
Function set:	NOT, MAX, IF-THEN-ELSE, AND, +, -, *, <, READ, WRITE, and PROG2
Indexed Memory:	128 integers.
Wrapper:	$\{0 \leq \mathit{MoveDown} \leq 18\}$ $\{19 \leq \mathit{MoveLeft} \leq 36\}$ $\{37 \leq \mathit{MoveRight} \leq 54\}$ $\{55 \leq \mathit{MoveUp} \leq 73\}$ $\{74 \leq \mathit{RotatePiece} \leq 91\}$ $\{92 \leq \mathit{Halt} \leq 109\}$ $\{110 \leq \mathit{DoNothing} \leq 127\}$
Fitness cases:	Between 3 and 40 games of Tetris. See Section 14.3.2.
Raw fitness:	The sum of the scores earned playing Tetris games: one point per block filled minus one point per hole.
Parameters:	Generations = 171, pop size = 1000, 5-member tournament selection.
Identification of best of run:	Each best-of-generation individual is validated over 200 unseen cases, and the individual with the best validation is deemed best-of-run. (The best-of-run individuals are then revalidated over more unseen games.)

1992], with the addition of READ and WRITE operations to access indexed memory. The indexed memory is an array of 128 integer values, each between 0 and 127. All functions are constrained to return values between 0 and 127, so the return value of an entire parse tree is always within that range.

Available parse tree terminals include the current piece type (PIECE-TYPE), which has seven possible values (see Figure 14.2). These values are evenly distributed across the range of possible values (0 through 127). Since random constants are selected from the same range, this distribution of PIECE-TYPE's values has a potential advantage over assigning

PIECE-TYPE the values zero through six: comparisons ("<") to random constants will be more likely to discriminate among the seven possible PIECE-TYPE values. Another terminal is the current piece's rotational orientation (PIECE-PHASE), which has four possible values that are similarly distributed. Five terminals are boolean flags: NEW-PIECE, which is on momentarily when a new piece appears, and four other flags indicating if the sensor is at the board's boundaries.

Evolved algorithms need a means to access an array of input data. In our approach to Tetris, we use a data access primitive akin to a "mobile sensor." The program has access only to one small piece of data at a time, and can incrementally move the sensor to access other pieces of data. The sensor's size is two by two, and these four bits compose the value encoded in the SENSOR terminal (between 0 and 15). For each piece, the sensor starts at the lower-left corner of the board.

Structured access to data has been used previously in genetic programming. For example, programs evolved to perform optical character recognition can operate by shifting templates (that are themselves evolved) around the input array [Andre 1994a]. Omega loops in DNA strings have been detected by evolved programs that access the data in a more structured way: the evolved program is forced to scan the entire input array exactly one time [Koza 1994; Koza and Andre 1996]. Similarly, primitive operations that iterate over a vector of data have been used [Andre 1994b; Handley 1995].

The return value of the program tree is mapped, according to the wrapper shown in Table 14.1, to one of the following actions: move the sensor up, down, left or right; rotate the piece; halt (immediately dropping the piece); or do nothing. When the program issues the *Halt* action, or a maximum number of program tree evaluations is reached, the sensor's position determines the horizontal location from which the current piece is dropped.

14.3.2 Fitness Cases

Each game of Tetris is a fitness case, uniquely determined by the particular series of pieces that appear at the top of the board. To implement this, each fitness case is represented as seed to a random number generator. The sequence of numbers resulting from a seed determines the sequence of pieces that compose a game.

Since the performance of an evolved program can vary between games, a successful program must perform well on average. It is not satisfactory for an evolved program to be proficient at only a small number of fitness cases – that is considered overfitting [Weiss and Kulikowski 1991]. Therefore, to measure the fitness of an evolved program accurately, it must be evaluated over a number of fitness cases. Since accuracy is improved by increasing the number of fitness cases, accuracy can be very expensive. Note that, despite increased fitness accuracy, the general performance of best-of-run individuals must be evaluated over

unseen test cases.

We use a *progressive fitness measure* that attempts to spend less time on individuals with lower fitness. Better individuals are tested over more training examples, and an individual's fitness is the sum of its performance measurements over the examples. To do this, the population is evaluated in phases, each consisting of I fitness cases. After each phase, a certain fraction of the population is considered "done," and is not evaluated further ("frozen") during the remaining phases. In this work, $I = 3$, and below-average individuals are frozen at each phase. These values were determined largely so that runs took less than 15 hours each. As a result, each evolved program is evaluated over three to 40 games. The progressive fitness measure is based on suggestions made by Teller [Teller 1994c] and Singleton [Singelton 1994]. However, a pilot study comparing the progressive fitness measure to playing only one game per fitness measure showed no significant improvement in performance. This may be a result of the fact that all runs converge on a relatively simple local minimum, as described in Section 14.4.

14.3.3 Why Tetris?

When playing Tetris, some circumstances require more sensor movements than others to process. Specifically, when a new piece appears at the top of the game board, an evolved program must scan the game board with the mobile sensor and determine the position from which to drop the piece. The number of times that the board sensor needs to be moved by the evolved program depends on the contents of the board and the type of piece being dropped. Therefore, Tetris is a viable test problem for dynamic computation time allocation.

Our approach to Tetris provides the genetic algorithm with limited domain knowledge. This is in contrast with, for example, optimizing a function that evaluates the game board's contents, as suggested by Parlante [Parlante 1994]. In that approach, each possible horizontal and rotational configuration for each piece is tried, and the one that results in the board configuration with the highest value, as determined by the board evaluation function, is selected. This approach incorporates the result of dropping a piece from all possible positions, and may therefore be better for solving Tetris. In real world applications, however, assessing and incorporating domain knowledge is often expensive or impractical [Angeline 1994].

In our approach to Tetris, the program must examine the game board for a suitable place to drop each piece, and is not permitted to try dropping it in more than one place. This problem allows no straightforward solutions and is more complex than classic genetic programming benchmarks. Like the Tartarus domain, indexed memory is important, since with no access to indexed memory, it is impossible for evolved programs to keep track

of the position of the sensor. In a pilot study, performance dropped significantly when memory was not made available to evolved programs.

14.4 Strategies that Result with No Time Pressure

All the programs we have examined that were evolved with no fitness pressure to optimize execution time follow a simple strategy: they horizontally distribute the pieces based on their piece type, with little or no sensitivity to the board's contents. That is, all the pieces of one type are stacked at a particular horizontal location on the board – each type filling a small number of columns. Additionally, some pieces are rotated before they are dropped, which improves this limited strategy. On average, less than one line is cleared per game. However, we have seen much more successful evolved programs that follow more elaborate strategies on variations of Tetris, as detailed below in Section 14.8.

The Tetris players evolved with no fitness pressure to optimize execution time do not use their allotted cycles efficiently. After moving the sensor to the horizontal position corresponding to the current piece type, resulting programs often expend the remainder of their allotted time "waiting." That is, instead of making use of the available halt operation to explicitly indicate that the sensor is at its horizontal destination, they avoid moving the sensor horizontally until time-out occurs by moving it vertically, by rotating the piece, or by issuing the *DoNothing* command. Since computation time is not incorporated into the fitness measure, their fitness does not suffer from this poor use of computation time. The use of computation time would be more efficient if such an evolved program dropped each piece as soon as it had achieved the appropriate rotational orientation and horizontal position.

14.5 Aggregate Computation Time Ceiling

In complex problem domains such as Tetris, some fitness cases require only a small number of operations to select an appropriate response, but others require more time to process. To use computation time efficiently, evolved programs must dynamically allocate computation cycles among such fitness cases. However, a program evolved using a fitness measure that does not consider execution time as a criterion can use time inefficiently with no penalty on its fitness. A method to improve computation time allocation must encourage an evolved program to make quick decisions, but should not prohibit it from spending more time in some situations, as required.

To improve the computation time allocation of evolved programs, we impose an *aggregate computation time ceiling* that applies over a series of fitness cases. Under this

condition, evolved programs must strategically allocate their allotted computation time in order to maximize fitness. A program that always spends a long time on each fitness case will get a low fitness measurement because it will be evaluated over fewer fitness cases before reaching the time ceiling. On the other hand, a program that always spends a short time on each fitness case will presumably not be able to develop a strategy that is complex enough for success. Successful programs must strategically allocate their computation time, spending less time on some cases and more time on others.

It is not a foregone conclusion that evolved programs will be able to dynamically allocate their computation time since this would require them to spontaneously halt. If an evolved program never issues the *Halt* operation, it will be evaluated over fewer fitness cases before reaching the aggregate time limit, so its fitness will suffer. An evolved program must solve the problem of when to halt in addition to the domain problem over which it is evolved. This difficulty is partially alleviated by maintaining a time limit on each fitness case in addition to the aggregate limit over a series of fitness cases. All that is required of such an evolved program is that it explicitly halt for *some* cases. With no time limit placed on individual fitness cases, successful individuals must explicitly halt for *all* cases.

With an aggregate time ceiling that is too low, it is impossible to evolve optimal strategies. In this case, individuals will be forced to make tradeoffs to strategically allocate their allotted time. For example, an evolved program could make a suboptimal choice to save time for fitness cases that demand more computation time.

The dynamic use of resources other than computation time may also result from placing an aggregate limit over a series of fitness cases. Example resource use includes memory allocation or fuel consumption. By the same arguments, the optimal consumption of any such resource is dynamic from one test case to the next, and an aggregate consumption limit imposed over a series of fitness cases could elicit dynamic resource allocation. Note that the aggregate constraint can be eliminated when validating evolved programs. This sometimes allows for improved performance, since evolved individuals may make efficient use of a larger amount of the resource.

Considering the execution time of evolved programs as a criterion in addition to domain performance presents a multi-objective problem. Previous approaches to multi-objective optimization include weighting the multiple objectives and combining them into one fitness measure. For example, as an attempt to increase the parsimony of evolved programs, the number of points in evolved expression trees can be incorporated into the fitness measure [Koza 1992]. This approach has been expanded upon by incorporating Minimum Description Length principles into the fitness measure of evolved decision trees [Iba et al. 1994] and neural networks [Zhang and Muehlenbein 1995]. Other approaches avoid the need to combine the objectives into one measure by performing comparisons along multiple dimensions [Schaffer 1985; Fonseca and Fleming 1993; Horn et al. 1994; Ryan 1994].

Table 14.2
Applying the aggregate computation time ceiling to Tetris.

Concept	As applied to Tetris
fitness	score
fitness case	piece
series of fitness cases	game
fitness case time ceiling	piece time
aggregate computation time ceiling	game time

The aggregate computation time ceiling differs from previous approaches to multi-objective optimization in two fundamental ways. First, it places an absolute constraint on one of the objectives - time. This means that programs evolved under this constraint *must* spend less average time per fitness case than those without the constraint or they will not be allowed to complete a series of fitness cases. Standard multi-objective techniques penalize evolved programs for taking longer, but do not impose an absolute limit. Second, an evolved program that adheres to the time constraint can only improve its fitness by improving its domain performance. That is, such a program cannot further improve its fitness by decreasing its execution time. Other multi-objective approaches allow evolved programs to arbitrarily trade domain performance for computation time.

14.6 Applying the Aggregate Computation Time Ceiling to Tetris

The nature of the fitness function for Tetris is fairly complicated and deserves reiteration here. It comprises three distinct levels. First, a single evaluation of the evolved program tree is performed. Each such evaluation results in an action – either a movement of the roving sensor, a *Halt*, or a *DoNothing*. Second, after a number of actions, the current piece falls, and a new piece appears. At this point the indexed memory array is erased, breaking the continuity between pieces; we may thus consider each piece as a distinct execution of the program. That is, for the purposes of computation time allocation experiments, we consider each piece as analogous to a fitness case. Third, the game score is computed over a series of pieces; therefore, each game is analogous to a series of fitness cases. These concepts are summarized in Table 14.2.

To impose an aggregate time ceiling on programs evolved to play Tetris, we enforce a time limit on each game, the *game time*, that applies over the series of pieces composing a game. The aggregate computation time ceiling can be applied to other domains like Tetris in which there are numerous discrete subproblems (i.e. pieces). Further, this method can be applied to any problem with numerous fitness cases. In order to emphasize this latter point, we consider each piece a "fitness case." However, this is merely an analogy, since

domain performance is measured per game, not per piece.

With a game time ceiling imposed, a successful Tetris player must be proficient at the game, and yet on average spend less than the maximum time allowed per piece. If a program spends too long on each piece, the game time ceiling will expire before it has completed playing a game. However, if a program always spends too little time on a piece, it will not be able to select an appropriate position from which to drop the piece.

When applied to Tetris the aggregate time ceiling has no effect on the fitness measure until evolved programs have reached a certain remedial level of ability. This is because evolved programs with low fitness lose Tetris games (by stacking the first few pieces too high) before reaching the aggregate time limit. Therefore, the genetic algorithm can generate individuals with remedial abilities before it is necessary to create individuals that halt spontaneously.

The problem of how to define computation time has been addressed in several different ways. One method is to treat each program tree evaluation as a single time unit [Teller 1994a], or to assign each function a weight based on the number of CPU cycles it uses [Teller 1996]. In the present experiments, we consider each entire tree evaluation as a single time unit. Since each tree evaluation results in a single action (*MoveDown*, *MoveLeft*, *MoveRight*, *MoveUp*, *RotatePiece*, *Halt*, *DoNothing*), an individual that performs, for example, 13 movement actions before halting takes 13 time units.

The imposition of a fixed game time ceiling may seem unfair: as the population evolves, one would expect strategies to become more successful at Tetris, fitting more pieces on the board, and thus requiring more time to carry out. A fixed time ceiling would thus emphasize speed to the exclusion of higher-order strategies. However, in the present experiments, we compare the aggregate computation time ceiling to results with no aggregate time constraint. We will show that the aggregate computation time ceiling produces an improvement over this control experiment.

14.7 Method and Results

In this section we report the results of three sets of experiments. First, we tested a series of aggregate computation time ceilings. From these experiments, we chose an aggregate time ceiling deserving of further analysis, where the tradeoff between execution time and domain performance was most beneficial. In the second set of experiments, we compared the selected aggregate time ceiling to a simple, per-piece time ceiling – we have predicted that the former would result in more dynamic time allocation. Finally, we performed two additional control experiments.

The results in this section are based on the best-of-run individuals resulting from a batch

of runs with identical parameters, as measured after 172 generations; e.g., results for a batch with 23 runs is based on 23 best-of-run individuals. Best-of-generation individuals are validated over 200 unseen games of Tetris, and the individual with the best validation is deemed best-of-run. This best-of-run individual is then re-evaluated over a set of games that are common across all batches, but are unseen during the runs. This re-evaluation avoids the bias introduced by selecting from a group of 172 best-of-generation individuals. When evaluating, no game time ceiling is in effect, so games only end according to the rules of Tetris.

We use three measures of performance to evaluate resulting Tetris players. *Score* is the game score; the sum of the scores across a series of games composes the fitness measure. *Computation time* is the average time spent on each fitness case, i.e. the number of times the mobile sensor moved on average between the introduction of each piece and either a timeout or an explicit *Halt* command. *Efficiency*, defined as the ratio *(score / game duration)*, measures the rate at which evolved programs gain points while playing Tetris. Computation time and efficiency are purely analytic measures and play no role in selection or the fitness measure.

14.7.1 Imposing and Lowering an Aggregate Computation Time Ceiling

Our primary line of investigation involves comparing batches with a piece time of 13, but differing aggregate computation time ceilings. Starting with a batch of runs with unlimited game time (the control batch), we systematically lowered the game time ceiling from 240 to 60.

Figures 14.3 and 14.4 show the resulting average efficiency and average computation time, respectively, as a function of game time, and Table 14.3 summarizes these results, as well as indicating the number of runs performed with each time ceiling. One-factor ANOVAs (Analysis of Variance) for both score and computation time showed a main effect for game time. As predicted, imposing a game time ceiling caused the average time spent per piece to drop. Unfortunately, the average score decreased as well (from 73.4 to as low as 46.3). However, computation time decreased more drastically than score, as illustrated by the steady rise of efficiency in Figure 14.3.

There is a point at which efficiency rises with only a slight corresponding drop in score. Lowering the aggregate computation time ceiling from 180 to 120 yielded a sharp rise in average efficiency (from 0.294 to 0.376) while the average score fell only slightly (from 60.6 to 59.1). In the next set of experiments we further analyze this “hinge point.”

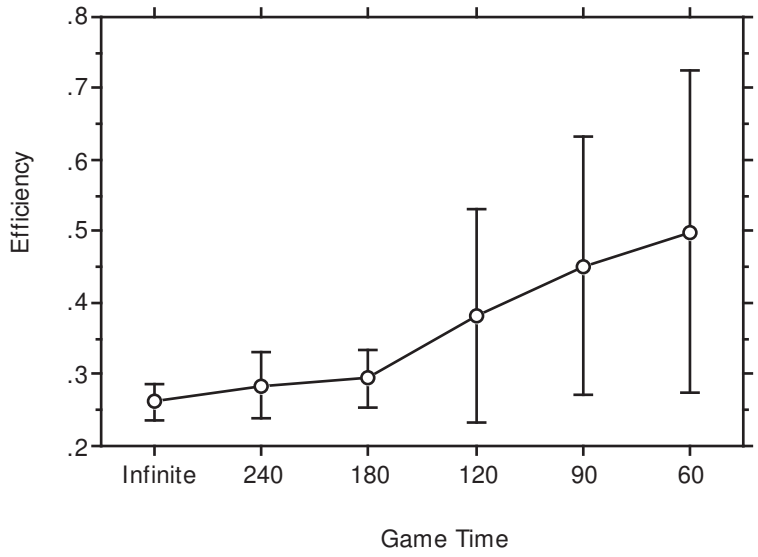

Figure 14.3
Mean efficiency as a function of game time ceiling (with standard deviation bars).

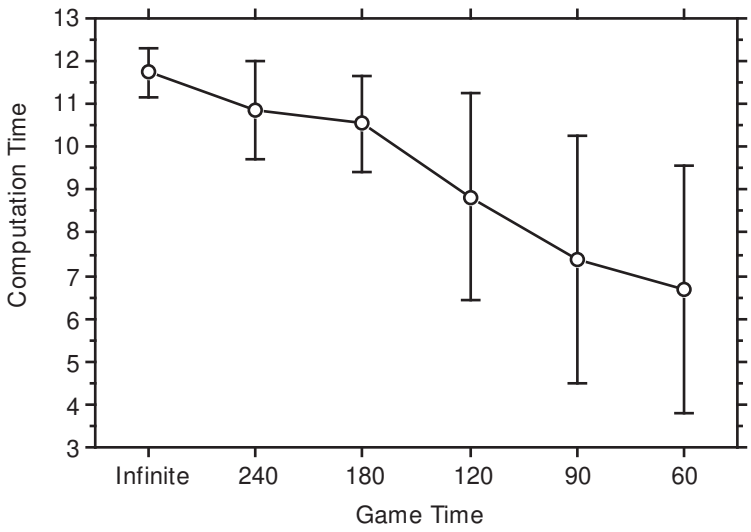

Figure 14.4
Mean computation time as a function of game time ceiling (with standard deviation bars).

Table 14.3
Means for score, computation time and efficiency with varying aggregate computation time ceiling.

Aggregate Computation Time Ceiling	Runs	Score	Computation Time	Efficiency
∞	26	73.413	11.744	0.263
240	25	67.177	10.850	0.285
180	12	60.558	10.552	0.294
120	**23**	**59.130**	**8.843**	**0.376**
90	12	52.201	7.368	0.452
60	8	46.346	6.672	0.500

14.7.2 Evaluating the Aggregate Computation Time Ceiling

We have described a method for improving the efficiency of resource use by imposing a limit on the use of that resource over a series of fitness cases. A more obvious way to decrease resource use is to decrease the limit applied to each fitness case. We have claimed that the former method is preferable because it encourages the evolved programs to use the resource *dynamically* – to spend less of the resource when possible in order to conserve it for cases in which more of it is needed. In this section we examine this claim.

To evaluate the aggregate time ceiling, we examine the "hinge point" batch of the previous experiment, with a game time ceiling of 120 (bold font in Table 14.3). We call this "Batch H." Since this batch had an average computation time of 8.843 (program evaluations), we will compare it to a batch of runs with a per-piece time ceiling of 9, and *no* aggregate computation time ceiling. We call this "Batch S" (for "simple").

Runs with the aggregate ceiling produced inferior results *on average*. As shown in Table 14.4, the mean score for Batch H is significantly lower than the mean for Batch S (59.1 vs. 71.3, t=-2.678, P<.01).

However, some runs using the aggregate computation time ceiling produced more favorable results. If we select the best-of-run individual with highest efficiency from each batch, also shown in Table 14.4, we see that Batch H managed to produce an individual with higher efficiency than the most efficient individual from Batch S. This individual also has a respectable average score (67.5).

Although a single data point is inconclusive, further analysis supports our intuition. Imposing the aggregate computation time ceiling increased the *variance* of efficiency (0.133 vs. 0.038). As a result, approximately one fourth of the runs in Batch H (6 of 23) resulted in an individual with higher efficiency than the most efficient individual from Batch S (which had 20 runs).

Since efficiency measures an arbitrary tradeoff between score and computation time,

Table 14.4
Lowering the per-case time ceiling (Batch S) vs. an aggregate computation time ceiling (Batch H).

	Aggregate Computation Time Ceiling	Piece Time	Most Efficient Individual			Average Score
			Efficiency	Score	Time	
Batch S	∞	9	0.442	73.465	7.40	71.285
Batch H	120	13	0.691	67.500	4.87	59.130

these results can be analyzed in further detail. For Batch S, computation time stayed close to the ceiling of 9, with a mean of 8.066 and a standard deviation of only 0.405. However, in Batch H, average computation times varied widely, from 11.52 down to 4.79, with a standard deviation of 2.427. Furthermore, a certain portion of the fastest individuals also gained high scores. The correlation coefficient comparing time and score was 0.172. Since this is close to zero, it is nearly as likely for score to be high and time to be low as any other relationship between the two.

How do we explain the fact that imposing an aggregate time ceiling produced good individuals, but only in some of the experimental runs? We surmise that the aggregate ceiling poses a more difficult problem for genetic induction. While is it difficult enough that some runs get mired in particularly suboptimal solutions, an occasional trial produces an individual that is both reasonably fast and reasonably fit. In many applications, one successful individual is all you need.

These results show that even though the genetic algorithm selects only for domain performance, imposing an aggregate computation time ceiling produces some best-of-run individuals that are fast. Further, encouraging dynamic time allocation is preferable to simply lowering the per-piece time limit since more efficient individuals can be produced.

14.7.3 Selecting a Per-Piece Time Ceiling for Tetris

All of the batches with an aggregate time ceiling also had a per-piece time ceiling of 13 in effect. Thus even when the program did not explicitly halt, the piece was dropped automatically after only 13 time steps. This seems to run counter to the spirit of the aggregate time ceiling: shouldn't the evolved programs be able to explicitly halt at the appropriate time in every situation?

Actually, no. All that is required for evolved programs to decrease their execution time is that they explicitly halt for *some* cases. When playing a game with no limit imposed for each game piece, successful individuals must explicitly halt for *all* pieces. This makes the solution more difficult, since it must not only calculate the proper position for each piece, but recognize when it has reached that position and issue the *Halt* response code. Also, since early partial solutions cannot rely on an automatic time-out, the genetic search may

Table 14.5
Performance suffers if no piece time is imposed.

Game Time	Piece Time	Score	Computation Time	Efficiency
120	∞	33.062	5.578	0.526
120	13	59.130	8.843	0.376

be less successful. Thus the use of a per-test case time limit "smooths" the search space, providing a hint for the genetic algorithm.

To verify this hypothesis, we ran a batch with an aggregate time ceiling of 120, but no piece-time ceiling. As shown in Table 14.5, the score suffered significantly as compared to a batch with a piece time of 13 ($t=6.307$, $P<.01$). However, the efficiency was significantly greater. This can be explained by the fact that, in order to get any points at all before game time ran out, the most successful individuals were the ones that converged on a strategy of halting almost immediately. The genetic algorithm could not escape this local minimum.

This leads to the question, is 13 too low a piece time for Tetris? The ideal per-piece time limit would not be so low as to preclude an optimal solution. We performed an experimental batch with piece time 26 and found a *decrease* in score and an *increase* in computation time. This reinforces our use of a per-piece limit of 13, but further analysis will be necessary to understand this decrease in performance.

14.8 Variations on the Test Problem

Evolved programs are more successful at variations of Tetris in which there is a smaller set of piece types. The most extreme example is Unis, for which each piece is composed of only one block. Since there is only one possible piece type, there is only one possible sequence of pieces, and therefore only one fitness case. Therefore it is not necessary nor possible to validate fitness over unseen Unis games. Many runs quickly find a simple strategy that fills the board row by row, and therefore clears lines forever.

Tritris is the variation of Tetris in which each piece is composed of three blocks, which results in two possible piece types (since all symmetries are accomplished by rotation). Resulting Tritris players are more successful than the Tetris players reported in this chapter. This improvement arises because evolved Tritris players successfully utilize both the piece type and the board contents when positioning a piece. The resulting strategies obtained for Tritris are difficult to describe formally, and vary widely from run to run. The experiments with Tritris vary significantly from those detailed in this chapter; for example, they follow a steady state model [Syswerda 1989]. They will be the subject of a future paper.

14.9 Conclusions and Future Work

Many new domains for genetic programming require evolved programs to be executed for longer amounts of time. For many such applications it is likely that some test cases optimally require more computation cycles than others. Therefore, programs must dynamically allocate cycles among test cases in order to use computation time efficiently.

Imposing an aggregate computation time ceiling over a series of fitness cases introduces a constraint on the behavior of evolved programs. Under this time constraint, an evolved program must take less time per test case on average. As long as a program does not take too much time on average, it is not penalized for taking greater than average time on fitness cases that require more time.

With an aggregate time ceiling in effect, resulting Tetris players allocate computation time more efficiently than those evolved with no such time constraint. The best runs with an aggregate time ceiling are superior to the best runs with no aggregate ceiling and less time per test case. This comparison illustrates the advantage of the aggregate time ceiling: more time can be spent on a test case when necessary, but spending less time, whenever possible, is encouraged. The aggregate computation time ceiling may be especially promising for domains in which the fitness cases vary in difficulty from one to another.

An evolved program must learn to spontaneously halt in order to vary its execution time among fitness cases. This is a problem it must solve in addition to the domain over which it is evolved. This difficulty is partially alleviated by maintaining a time limit on each fitness case in addition to the aggregate time limit. In this case, evolved programs need only issue a *Halt* command for *some* fitness cases. As a result, evolved Tetris players were much more successful with a time limit maintained on each fitness and test case.

In principle, an aggregate ceiling can be placed on the use of resources other than computation time, such as memory or fuel. Applying a limit on resource consumption over a series of fitness cases could encourage dynamic resource allocation. In this way, the present work may be extensible to optimizing the allocation of resources in general.

We are continuing our research in a number of lines to evolve more successful Tetris players. For example, preliminary results show that competitively co-adapting training cases [Hillis 1992; Angeline and Pollack 1993; Siegel 1994] increases the performance of evolved Tritris players.

Acknowledgements

Many people have provided valuable insights for this work, including Astro Teller, Peter Angeline, David Schaffer, Larry Eshelman, Kathy McKeown, John Koza, and Andy Sin-

gleton. Thanks also to Walter Alden Tackett and Aviram Carmi for the use of their SGPC software. Thanks to Nelson Minar, Jacques Robin, and David Andre for comments on an earlier draft of this chapter.

Research by Eric Siegel was partially supported by the Advanced Research Projects Agency and the Office of Naval Research under contract N00014-89-J-1782, and by the National Science Foundation under contract GER-90-24069, and was under the auspices of the Columbia University CAT in High Performance Computing and Communications in Healthcare, a New York State Center for Advanced Technology supported by the New York State Science and Technology Foundation.

Any opinions, findings, conclusions or recommendations expressed in this publication are those of the authors and do not necessarily reflect the views of the New York State Science and Technology Foundation.

Bibliography

Andre, D. (1994a). Automatically defined features: the simultaneous evolution of 2-dimensional feature detectors and an algorithm for using them. In Kinnear, K., editor, *Advances in Genetic Programming*. MIT Press, Cambridge, MA.

Andre, D. (1994b). Learning and upgrading rules for an OCR system using genetic programming. In *Proceedings of the First IEEE Conference on Evolutionary Computation*, Piscataway, NJ. IEEE.

Angeline, P. (1994). Genetic programming and emergent intelligence. In Kinnear, K., editor, *Advances in Genetic Programming*. MIT Press, Cambridge, MA.

Angeline, P. and Pollack, J. (1993). Competitive environments evolve better solutions for complex tasks. In *Proceedings of the Fifth International Conference on Genetic Algorithms*, San Mateo, CA. Morgan Kaufmann.

Fonseca, C. and Fleming, P. (1993). Genetic algorithms for multiobjective optimization: formulation, discussion and generalization. In *Proceedings of the Fifth International Conference on Genetic Algorithms*, San Mateo, CA. Morgan Kaufmann.

Handley, S. (1995). Classifying nucleic acid sub-sequences as introns or exons using genetic programming. In *Third International Conference on Intelligent Systems for Molecular Biology*, Cambridge, UH. AAAI Press.

Harris, C. (1996). Compiling genetic programs: a problem-independent genetic programming system. Technical report. Submitted to Genetic Programming 96.

Hillis, D. (1992). Co-evolving parasites improves simulated evolution as an optimization procedure. In C. Langton, C. Taylor, J. F. and Rasmussen, S., editors, *Artificial Life II*, Reading, MA. Addison-Wesley Publishing Company, Inc.

Horn, J., Nafpliotis, N., and Goldberg, D. (1994). A niched pareto genetic algorithm for multiobjective optimization. In *Proceedings of the First IEEE Conference on Evolutionary Computation*, Piscataway, NJ. IEEE.

Iba, H., de Garis, H., and Sato, T. (1994). Genetic programming using a minimum description length principle. In Kinnear, K., editor, *Advances in Genetic Programming*. MIT Press, Cambridge, MA.

Koza, J. (1992). *Genetic Programming: On the programming of computers by means of natural selection*. MIT Press, Cambridge, MA.

Koza, J. (1994). *Genetic Programming II: Automatic Discovery of Reusable Programs*. MIT Press, Cambridge, MA.

Koza, J. and Andre, D. (1996). Locating transmembrane domains in proteins using genetic programming. In Angeline, P. and Kinnear, K., editors, *Advances in Genetic Programming 2*. MIT Press, Cambridge, MA. Chapter 8 in this volume.

Langdon, W. B. (1994). Quick intro to simple-gp.c. Internal Note IN/95/2, University College London, Gower Street, London WC1E 6BT, UK.

Maxwell, S. (1994). Experiments with a coroutine execution model for genetic programming. In *Proceedings of the First IEEE Conference on Evolutionary Computation*, Piscataway, NJ. IEEE.

Parlante, N. (1994). Personal communication.

Ray, T. S. (1991). An approach to the synthesis of life. In Langton, C., Taylor, C., Farmer, D., and Rasmussen, S., editors, *Artificial Life II*. Addison-Wesley.

Ryan, C. (1994). Pygmies and civil servants. In Kinnear, K., editor, *Advances in Genetic Programming*. MIT Press, Cambridge, MA.

Schaffer, J. (1985). Multiple objective optimization with vector evaluated genetic algorithms. In Grefenstette, J., editor, *Proceedings of the [First] International Conference on Genetic Algorithms*. Lawrence Erlbaum.

Siegel, E. (1994). Competitively evolving decision trees against fixed training cases for natural language processing. In Kinnear, K., editor, *Advances in Genetic Programming*. MIT Press, Cambridge, MA.

Singelton, A. (1994). Personal communication.

Syswerda, G. (1989). Uniform crossover in genetic algorithms. In Schaffer, J., editor, *Proceedings of the Third International Conference on Genetic Algorithms*. Morgan Kaufmann.

Teller, A. (1994a). The evolution of mental models. In Kinnear, K., editor, *Advances in Genetic Programming*. MIT Press, Cambridge, MA.

Teller, A. (1994b). Genetic programming, indexed memory, the halting problem, and other curiosities. In *Proceedings for the 7th Annual Florida AI Research Symposium*.

Teller, A. (1994c). Personal communication.

Teller, A. (1994d). Turing completeness in the language of genetic programming with indexed memory. In *Proceedings of the First IEEE Conference on Evolutionary Computation*, Piscataway, NJ. IEEE.

Teller, A. (1996). Evolving programmers: the co-evolution of intelligent recombination operators. In Angeline, P. and Kinnear, K., editors, *Advances in Genetic Programming II*. MIT Press, Cambridge, MA. Chapter 3 in this volume.

Weiss, S. and Kulikowski, C. (1991). *Computer Systems that Learn*. Morgan Kaufmann, San Mateo, CA.

Zhang, B.-T. and Muehlenbein, H. (1995). Mdl-based fitness functions for learning parsimonious programs. In Siegel, E. and Koza, J., editors, *Genetic Programming: Papers from the 1995 Fall Symposium*. AAAI Technical Report FS-95-01.