Evolving a Program to Play the Game Minesweeper

Chris Quartetti
Stanford University
April 1998
http://quartetti.net/chris

ABSTRACT: This paper describes how genetic programming was used to evolve a program to play the game Minesweeper. There are several levels of strategy that can be used in this game and the goal is to evolve a program that displays as many of these as possible. An ideal solution will perform well on cases which it was not tested, specifically varying board size and number of mines. This is a "clean-hands" approach in which no complex domain-specific functions nor primed individuals are given to the GP at the outset.

1. Introduction

Genetic programming is a product of the genetic algorithm developed by John Holland (1975). Programs can be evolved using a model of Darwinian natural selection and survival of the fittest. The benefit of this approach is it can opportunistically explore multidimensional search spaces and remains very robust with regard to deceptions (local maxima/minima). The game Minesweeper was chosen because it is a simple but challenging game-- a human player can employ increasingly complex strategies to improve his chance of success. Because it is not a complete knowledge game, there are often times when the player must make a move whose success is based on weighing a series of probabilities. Hence, the better a player can calculate these probabilities from the current state of the game board, the higher the chance of success.

2. Statement of Problem

Minesweeper is a one-player game, played on a 8×8 grid of covered squares, 10 of which contain mines, randomly placed at the start of the game. The goal is to find all the mines as quickly as possible. Uncovering a square reveals a number representing the number of mines on adjacent squares (this number can range from zero to 8). If the player successfully uncovers all the squares without mines, the clock stops and the game is over. If a square containing a mine is uncovered, it blows up and the game ends in a loss. The player can optionally mark squares to help remember where he thinks the mines are. The game can be played at more difficult levels which involve larger grids and more mines. In this paper, a subset of the game will be used as a fitness measure due to CPU time and memory limitations. In general, the number of possible board configurations is given by the binomial theorem:

$$\binom{squares}{mines} = \frac{squares!}{mines!(squares - mines)!}$$

Hence the standard 8×8 grid with 10 mines has over 4 billion starting configurations. Of course this can be reduced if one rotates the board to see similar cases. A vastly larger reduction can be obtained by generalizing the relation of covered squares to the numbers representing surrounding mines.

3. Methods

The following table summarizes the key decisions for the run discussed in this section:

Table 1 Tableau

Objective	Find a program to play Minesweeper on a 6×6 board with 1 mine.
Terminal set	CONST (random constants 0-8)
Function set	MOV UNC MRK UNMRK PROGN IFCOV IF ADD SUB EQ GT LT AND OR NOT NUM ADF0 ADF1 ADF2
Architecture of program	One RPB which can call 3 ADFs, taking 3, 2, and 1 parameters. ADFs can call any other function except themselves and ADFs taking fewer parameters (to prevent recursion).
Fitness cases	36 (all possible configurations of 1 mine on a 6×6 board)
Raw fitness	Sum (taken over all fitness cases) of 2 times the number of unmined squares uncovered minus 1 if a mined square is uncovered. Maximum possible: 2449.
Standardized fitness	Maximum possible raw fitness minus raw fitness.
Adjusted fitness	$\frac{1}{1+standardizedFitness}$
Hits	Same as raw fitness.
Wrapper	None.
Parameters	M= 50,000, G= 151 reproduction 10% crossover 90% (10% leaves, 80% interior nodes) mutation 0%
Success predicate	Standard fitness of zero.
Result designation	Best-so-far individual.

Function and terminal sets:

The function set was chosen to be general, primitive operations to maintain a "clean-hands" approach to the problem. The operations that may not be self-explanatory are summarized in Table 2.

Table 2 Selected Functions and Terminals

Function mnemonic	**Purpose**
CONST	Integer constant with 9 possibilities (0-8) to represent eight different compass directions plus a null value.
MOV	Move around the grid. This is parameterized to accept a direction 0-7 representing all vertical, horizontal, and diagonal movements.
UNC	Uncover the square in the grid indicated by the direction parameter.
MRK UNMRK	Mark/unmark the square indicated by the direction parameter. An uncovered square may not be marked.
PROGN	No intrinsic function other than to hook two other functions together to

	build a larger program tree.
IFCOV	If the current square is covered, evaluate the first argument, otherwise evaluate the second.
NUM	Return the number of adjacent mines for the square indicated by the direction parameter.
ADFn	Automatically defined functions co-evolved with the main branch.

In order to see how programs might be constructed from the above primitives, two simple examples follow. The first is the simplest program to score any hits (the ADFs have been omitted because they are not used here):

```
RPB_Num 0
(UNC 0)
```

The RPB (Result Producing Branch, or main program branch) contains only one statement. This uncovers the first square on each grid giving a raw fitness of 69 out of 2449 (adjusted fitness 0.00042). See below for more information on fitness. The next example is more complex-- the following code uncovers all the squares on the first row of the grid then moves down to the next row:

```
RPB_Num 0
(PROGN (ADF0 0 0 0)
       (MOV 1))
ADF_Num 0
(PROGN (ADF1 0 0)
       (ADF1 0 0))
```

```
ADF_Num 1
(PROGN (ADF2 0)
       (PROGN (ADF2 0)
              (ADF2 0)))
ADF_Num 2
(PROGN (UNC 0)
       (MOV 3))
```

RPB0 simply calls ADF0 and moves down. ADF0 calls ADF1 twice, ADF1 calls ADF2 three times (amounting to the six cells in the row). ADF2 uncovers the current square and moves right. The raw fitness of this individual is 384 (adjusted fitness 0.00048).

Search Space

A great variety of programs can be constructed using the elements detailed above. Program branches (RPB and ADFs) are represented as tree structures, each limited to 200 nodes for the run detailed above. The number of possible programs that can be represented is difficult to estimate, but it is bound by the number of different terminals and non-terminals raised to the power of the number of nodes in the tree. This is akin to taking a string of length 200 of an alphabet of all the functions and terminals. In this case, it is $20^{200}+19^{200}+18^{200}+17^{200}= 1.6\times10^{260}$. It is admittedly a very high upper bound because it counts trees of less than maximum size many times, ex. the case where the first node is a terminal will be highly overcounted. It also counts trees that are invalid—that is the 200^{th} node may be a function instead of a terminal. A simple lower bound is given by the sum of the known possible trees. For example, the largest tree that can be built of 2 argument functions has 127 nodes (63 interior, 64 leaves). Each of the interior nodes can be one of 10 2-argument functions (for the RPB) or absent. Since there is only one type of leaf node, no additional possibilities arise here. This type of tree can be made $(10+1)^{63}= 4\times10^{65}$ different ways. Similarly, there are 1.2×10^{19} ways to build a tree with the available 3-argument functions for the RPB. Continuing for each of the ADFs, the lower bound for the 4 trees is 8.13×10^{65}. This does not take into account the possibility of different trees that produce the same results, but short of enumerating and testing each one, it is impossible to calculate.

Given these bounds, I believe a reasonable estimate of the actual search space is closer to the lower bound, perhaps 10^{68}.

Fitness

Fitness is evaluated over all possible 1-mine board configurations for two reasons: first to ensure a fully correct program is evolved, and second to find a program which will generalize over a larger playing grid. On a 6×6 grid with one mine, this only amounts to 36 cases.

The raw fitness of an individual program on a particular grid is the number of squares uncovered times two minus a penalty of one for blowing up. This small penalty was chosen to avoid the case in which taking no action has a higher fitness than actually trying something and possibly failing. This will cause the individual to keep trying until winning or blowing up, which fits well with the real game—there's no payoff in giving up. Total raw fitness is the sum over all test cases. Standardized fitness is calculated as *maxRaw – rawFitness* where *maxRaw* is the maximum raw score over all test cases. This maximum score is calculated as a perfect game on all grids except one, which allows for the unavoidable case in which the first square uncovered is a mine. The maximum possible raw fitness for the run outlined in Table 1 is (*#squares* - *#mines*) × 2 × (*#testCases* - *#mines*) - *#mines* = 35×2×35-1= 2449. If a subset of all possible test cases is used, the adjustment above for uncovering a mine on the first step is not used. Adjusted fitness is used to rank individuals in a population (Koza 1992): *adjusted fitness*= 1/(1 + *standardized fitness*). Adjusted fitness has the advantage of decreasing rapidly towards 1 as the individual approaches maximum fitness, which is not the case if a more traditional measure is used: 1-(*standardized fitness* / *maximum fitness*).

The element of elapsed time found in the real game is not present in the fitness measure, but each individual is given a fixed number of program node evaluations to finish, here 200 × squares= 7200 is used. There is no penalty for timing out.

A 6×6 board was chosen as it is sufficiently large that the 8 squares adjacent to a given square do not constitute a large portion of the board. In preliminary runs, a 4×4 grid was used, but the solution did not generalize well to larger boards. Likewise, 6×6 is not so large that covering the board without iteration operators requires an inordinate number of program steps or program nodes.

Genetic Operators

Only the crossover and reproduction operations are used in this problem. Crossover is applied selected individuals 90% of the time—this is split into 10% for leaves and 80% for interior nodes. When a particular individual is selected for leaf crossover, leafs nodes are randomly selected in two individuals and interchanged. A similar process is used for interior nodes. Any invalid structures generated during crossover are rejected and the operation is retried until a valid structure is generated or a maximum number of attempts is reached (10,000). If no valid structure can be made, the operation is changed to a simple reproduction.

These operators are sufficient to explore the entire search space for the following reasons:

- Trees can both grow and shrink by interchanging different sized subtrees via crossover at interior nodes.

- Any terminal can be interchanged with any other in the population via leaf node crossover.
- Any interior node can effectively be interchanged with any other. This happens when two subtrees are interchanged but the only differ at the node selected for crossover (or any one interior node).
- Any function or terminal can appear at any valid tree position in the initial population. These can be spread through the population on subsequent generations using the crossover operation.

Parameters

The parameters used in this run were chosen after some experimentation with smaller problems. Fifty or more generations were needed to solve most any non-trivial problem using populations in the 1,000 to 5,000 range. Many runs reached the maximum of 100 to 2,000 generations finding no solution, often hitting a plateau or making very slight increments every 20 to 600 generations. Given the amount of computer time this takes, it is worthwhile to use a larger population to cover more of the search space, and will often take fewer generations to find a solution. Population size 50,000 was chosen because it is the largest population that can be run in 128MB RAM without thrashing and still leave a little memory for other users (runs were conducted on shared resources). The tradeoff of a still larger population using virtual memory versus 50,000 in RAM was not worthwhile—run grew too long if available RAM was exceeded. If 50,000 is actually larger than needed, the only downside is that it may take longer to find a solution than using an optimum population. It is safer to err on the side of a large population because the run will succeed rather than needing to rerun a smaller population.

151 generations was chosen as a reasonable maximum because populations often start to plateau after this point. It usually makes runs short enough to get feedback on the other parameters so the scenario can be tweaked for future runs.

Reproduction, crossover, and mutation were set at the generally accepted numbers (Koza 1992) and not touched. Changes to the M, G, terminal/function sets, and fitness measures had a much greater effect on the outcome hence time was invested there.

Additional Details

The initial population is created randomly, ensuring the tree depth for each of the branches is between 1 and 5. No domain-specific information is used to prime individuals in the initial population, neither whole nor in part.

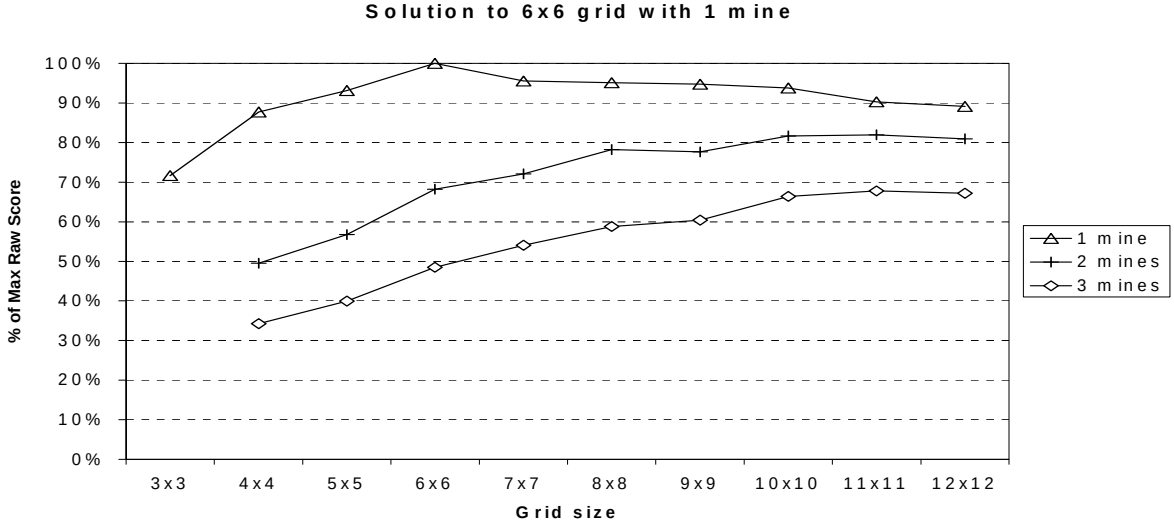
DGPC, Dave's Genetic Programming in C, (Andre 1994), was used to evolve programs. This was chosen because the GP engine is implemented in portable C code and functions called by the evolving individual are represented in C code which is compiled and linked into DGPC. DGPC is readily built on virtually any system with a C compiler and it has a great speed advantage over traditional interpreted systems, such as LISP.

4. Results

Figure 1 illustrates the performance of the best-of-run individual (run #542). The y-axis represents the percentage of the maximum raw score for the indicated configurations. The individual was tested on all possible boards with one mine and 100 random configurations with

two mines. Unfortunately it only scores 100% on the boards used as test cases during its evolution, but it does remain above 90% from 5×5 to 11×11 boards with one mine. The performance increase for larger boards with two mines can likely be explained by the greater likelihood of more space between mines, looking more like a single-mine environment locally.

Figure 1 Generalization of solution



This individual was well within the bounds of the maximum tree size of 200 with 162, 81, 52, and 12 nodes for the RPB and ADFs, respectively (307 total). The number of program steps needed for the 6×6 boards with 1 mine averaged 2,546, with the maximum 4,851 still well below the 7,200 allowed. Code for the 100% correct player for the 6×6 grid with 1 mine (raw fitness 2,449):

```
Num RPBS 1 Num ADFS 3 ARITY 3 2 1
RPB_Num 0

(LT (ADF2 (AND (ADF1 1 7)(ADF0 5 4 2)))(OR (IF (ADF1 5 7)(ADF2 8)(GT (ADF0 (ADD
(LT 1 5)(MRK 7))(ADD (LT 1 4)(ADF0 6 0 3))(ADD (PROGN 2 3)(PROGN 0 2)))(ADF0 (UNMRK (IF
(ADF1 5 7)(ADF2 8)(GT (ADF0 (ADD (LT 1 5)(MRK 7))(ADD (LT 1 4)(MRK (LT
(PROGN 0 8)(ADF0 7 0 0))))(ADD (PROGN 2 3)(ADF0 7 4 2)))(ADF0 (NOT 4)(GT (MRK (LT
(PROGN 0 8)(ADF0 5 4 2)))(ADF0 (ADF0 2 (NOT 4)(UNMRK 5))(ADF0 (LT
(PROGN 0 8)(ADF0 7 0 0))(AND 2 3)(ADF0 6 5 3))(EQ (MOV 8)(UNC 4))))(LT
(UNC 4)(PROGN 6 4))))))(PROGN (ADD (ADF0 8 0 6)(PROGN 2 3))(ADF0 (AND (SUB 0 6)(UNC 5)) 2
(LT (PROGN 0 8)(ADF0 7 0 0))))(LT (UNC 4)(MOV 8)))))(PROGN (IFCOV 1 4)(ADF0 7 7 7))))

ADF_Num 0

(ADF1 (OR (PROGN (ADF2 (PROGN (EQ (EQ (IFCOV 8 (ARG0))(EQ (ADF1 (MRK (ARG2))(ADF2 (MRK
(ARG2))))(UNC (OR (GT  7 3)(ADD (ARG0) 6)))))(UNMRK (MOV 1)))(UNC (ADF1 2 7))))(UNC (GT
(UNC (LT (ADD 8 4)(ADF1 (ARG2)(ARG1))))(ADD (NUM (ADF1 4 (ARG1)))(IF (NUM (MRK 3))(LT
(PROGN (EQ (MOV 8)(MOV 7))(ADD (ARG1) 5)) 3)(SUB (MOV (ADF2
(ARG1)))(UNC ARG1))))))))(ADF1 (LT 0 (ARG1))(ADD 0 4)))(ADF2 (PROGN (ARG2)(LT 1 4))))

ADF_Num 1

(IF (ADD (NOT 4)(ADF2 (MOV 7)))(LT (OR 0 8) 3)(IF (NUM (EQ 6 (ARG0)))(LT (PROGN (PROGN
(EQ (MOV 8)(MOV 7))(UNC 5))(UNC 2)) 3)(SUB (IFCOV (MOV (ARG0))(AND (SUB (UNC (ARG0))(UNC
(ARG0)))(OR (UNMRK (ARG0))(SUB (ADD 8 4)(UNC (MRK 5))))))(UNC (ARG1)))))

ADF_Num 2
```

```
(SUB (LT (IFCOV 8 (ARG0))(IF (ARG0) 3 (ARG0)))(UNC (MRK 5)))
```

There are some interesting points in the ancestors to the best-of-run above, starting with the best of generation (BOG) individual from generation 0, run #542 (107 nodes, raw fitness 745):

```
RPB_Num 0
(LT (ADF2 (AND (ADF1 1 7)
               (ADF0 5 4 2)))
    (OR (IF (ADF1 5 7)
            (ADF2 8)
            (PROGN 2 3))
        (PROGN (IFCOV 1 4) (ADF0 7 7 7))))

ADF_Num 0
(ADF1 (OR (PROGN (LT (ARG1) 2)
                 (LT 8 (ARG0)))
          (ADF1 (MOV (ARG2))
                (MRK (ARG0))))
      (ADF2 PROGN (ARG2)
                  (LT 3 (ARG2)))))
ADF_Num 1
(IF (ADD (UNC (ARG0))
         (ADF2 (MOV 7)))
    (OR (IF (PROGN (ARG0)(ARG1))
            (MOV (ARG1))
            (AND (ARG0)(ARG0)))
        (MOV (ADF2 (ARG1))))
    (IF (NUM (MRK 3))
        (LT (NOT (ARG1))
            3)
        (SUB (IF (ARG0)(ARG0)(ARG1))
             (UNC (ARG1)))))

ADF_Num 2
(IF (AND (SUB (UNC (ARG0))
              (UNC 2))
         (OR (UNMRK (ARG0))
             (ADD 4 (ARG0))))
    (PROGN (GT (ARG0)
               (IFCOV 7 (ARG0)))
           (ADD (ARG0)(UNMRK 5)))
    (NOT (PROGN (UNMRK 8)(ARG0))))
```

This individual employs a top-down approach to uncovering mines. It works only on the top 3 rows of the grid, always uncovering the exact same pattern of squares shown on Grid 1a (the '0's represent uncovered squares, 'c' covered, 'm' marked, '.' mine, 'M' marked mine).

Grid 1a	Grid 1b	Grid 1c	Grid 1d
m c c c 0 0	m c m **1** M **1**	**0** m **0** m **0** c	0 0 0 0 m 0
0 0 0 0 0 0	m 0 m **1** m **1**	m 0 **0** c **0** m	0 0 0 0 0 m
0 0 0 c 0 0	c 0 0 **0** m **0**	m 1 **0** 0 **m** 0	c 1 0 0 0 0
c c c c c c	**0** m 0 0 **0** 0	M **1** 0 0 m 0	**M** m 0 0 0 0
c c c c c c	**0** m 0 c **0** c	m **1** 0 **0** m **0**	1 1 0 0 0 0
c c c c c .	c c 0 m m 0	**0** **0** 0 **0** **0** 0	0 0 0 0 0 0

If no mines are uncovered, the program finishes and it moves on to the next board. The best of generation 2 (97 nodes, raw score of 969) also uses a pattern, but it is completely different, and likewise insensitive to the placement of mines (Grid 1b). Generation 3 (102 nodes, raw score 1104) is a variation on the pattern, still unaware of mines. The differences in uncovered squares between the two grids (1b and 1c) are shown in bold underline—moving from generation 2 to 3, 8 squares are lost, but 13 gained for a net gain of 5.

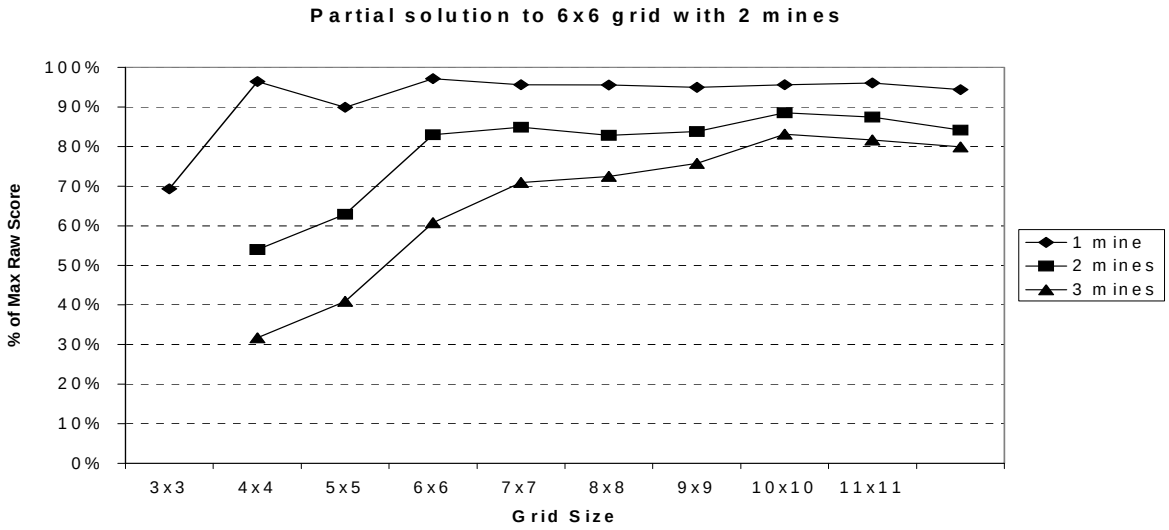
Succeeding generations continue to proceed with various patterns, switching from vertical to diagonal, apparently ignoring mines until generation 9. Generation 10 (149 nodes, raw score 1844) can actually steer clear of some mines in its path. In this case, it normally uncovers the square with the mine (highlighted), but in this case it leaves it covered, along with two of its neighbors (Grid 1d)

Skipping forward to the penultimate individual of the run, generation 66 (438 nodes, raw score 2447) is only one square shy of perfect fitness. It times out at 7200 steps on test 15. When GP finally solves the problem 4 generations later, test 15 is completed in 4097 time steps. This final change allows all 36 tests to run significantly faster, using 91,648 steps instead of 146,972, a 38% reduction. Unfortunately the differences in the code are not easy to characterize, amounting to several large changes including an almost completely new ADF2. The best-of-run has 307 nodes, 131 (30%) less than the previous best of generation.

Generalization

The complete solution to the 6×6 grid with 1 mine (run #542) generalized relatively well, however performance on grids with 2 mines was not terribly good, only climbing over 80% with a 10×10 grid. The example below is a partial solution evolved on a 6×6 grid with 2 mines (run #543/35, 419 nodes), tested on the same cases as Figure 1 for the graph. This shows much stronger, consistent performance on the 2 mine grids, scoring better on all grid sizes. In addition, the performance on 1 mine grids is as good as #542 in all cases except the 6×6 and 5×5 grids. Again, the performance scales up to larger grids better.

Figure 2 Generalization of partial solution to 6×6 grid with 2 mines



Interpretation

Run #542 follows what could be called a strategy if it were attributed to a human. First it works out getting around the board and uncovering squares. After it becomes somewhat proficient at this, it then turns its attention to trying not to uncover mines. These may well be two very large order schema working together, each contributing to fitness, but neither self sufficient. It is difficult to identify smaller schema without access to all individuals in a generation, and even so,

finding matching patterns in the code of 50,000 individuals is another problem in itself. Part of the difficulty in having a large population is several things are going on at once, and best of generation individuals can look quite a bit different even when separated by only one generation.

The two runs cited in the figures above show that the more robust solution is created with a more difficult set of test cases. Even though run #542 can play a perfect game on a 6×6 grid with one mine, its performance deteriorates when subjected to different configurations. Run #543 is actually incomplete—the individual tested is from generation 43 of a run 4 days in progress. It is part of a run of 50,000 individuals being tested on a 6×6 grid with 2 mines and 100 random test cases. Due to the large number of test cases, larger tree sizes, and a longer timeout, each generation is very time consuming to evaluate.

There appear to be three general stages to a run. First a very general solution is found that works to some extent on most test cases. Next all the test cases are improved upon roughly simultaneously. If an individual is taken from this point in the run and tested on new test cases, it will likely work well on the cases it has been evolved with, but score very low on the new ones. The final state is to perfect the test cases, and if enough cases are provided, also evolve more generalizable "skills".

A too-large population can slow progress sometimes, presumably because it takes more time for all the "good parts" to find one another and assemble themselves into one optimum individual.

Difficulties Encountered

Early experimental runs did not give good results for several reasons. The initial configuration attempted was an 8×8 grid with 10 mines. The first problem used far too few test cases—even 20 test cases is inadequate for a problem of this complexity. Also, the initial settings for maximum tree size were too small and hindered runs. 200 nodes per branch is acceptable for 6×6 grids with one mine, but problems with 2 or more mines require more nodes. Another problem was an inadequate timeout limit. An artificially low constraint here caused runs to last much longer than necessary. In fact, after doubling the timeout limit, a solution was obtained in a good deal fewer generations, and surprisingly, this solution required fewer timesteps to run than the previous solution. The solution to all these problems is "more"—more test cases, more timesteps, more tree nodes. Finally, some preliminary runs took much too long. Computer time and elapsed time could have been better spent running several small populations and using the results to tune the system for larger runs later.

The moral of this experience is to start small, as small as possible in fact. As runs produce results, analyze the intermediate and best-of-run individuals to see how close they are to limitations on size, run time, etc. and how well they generalize to other nearby test cases.

Replication of Results

The run to solve the 6×6 problem with one mine (population 50,000, solved by generation 70) took 47 hours on a Hewlett-Packard 9000 C180 workstation (PA8000 CPU, 128MB RAM). The majority of the time, DGPC was the only process running. DGPC version 2.0 was used as a basis, with the following augmentations: Node evaluations counted, call to "stopEval" function

added to Eval() to check fitness case evaluation termination criteria (exploded mines, all squares uncovered except mines, and timeouts). The code was built with g++ 2.7.2.2.

5. Future Work

A future goal is to evolve a fully generalizable player for grids up to 30×16 with 99 mines (this is "expert" mode in Minesweeper). Further analysis of results from the cited runs should lead to a revision of the function and terminal set. At this point, candidates for addition are as follows:

- Terminals that return the total number of mines on the board, the number of rows, and the number of columns (these will help scaling).
- Iteration function, implemented as a macro, accepting a numeric argument and the root of a subtree to iterate over.
- Load/store register functions.
- Controlled wrapping—in the runs cited above, any movement or query that moves off the edge of the board is wrapped back on the other side (toroidal playing board). This is not faithful to the original game and must be changed. This the root of some conflict in the game—the numbers representing adjacent mines do not count mines adjacent "over the edge".

After finishing Minesweeper, Tic-tac-toe is next. GP solutions to the game have been found, but they are not done with a "clean-hands" approach.

6. Conclusion

A perfect player can be evolved for a subset of the Minesweeper game without resorting to primed individuals or complex building blocks. The best-of-generation individual exhibits general abilities for both larger and smaller grids with more mines, but performance tapers off once removed too far from its area of expertise. A more robust individual was evolved using a larger number of more complex test cases. Even though this individual is far from a perfect solution for its test cases, it performs much better than the simpler "perfect" individual. Analyzing the test results for all the evolved individuals has shown skills ranging from flawed to acceptable compared with the most basic human strategy.

References

Andre, David 1994. *DGPC- Dave's Genetic Programming in C*. Software, unpublished.

Holland, John H. 1975. *Adaptation in Natural and Artificial Systems*. Ann Arbor, MI: University of Michigan Press.

Koza, John R. 1992. *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. Cambridge, MA: MIT Press