BrilliANT: An Entry for the GECCO'2007 Ant Wars Contest

Wojciech Jaśkowski, Krzysztof Krawiec, Bartosz Wieloch
Poznan University of Technology, Poznań, Poland

June 15, 2007

1 Motivations for our ant design

One of our first observations concerning Ant Wars was that ant's field of view (FOV) is rather small (24 fields when excluding the actual ant's position). If one considers food locations only and takes into account rotational invariance and symmetry, there are $2^{24}/4/2 = 2^{21} = 2097152$ unique FOV states (when ignoring the constraint placed on the amount of food and the existence of opponent). As FOV occupies approx. 20.7% of the ant's world and the total number of food pieces amounts to 15, the expected number of food pieces within FOV is slightly more than 3 when the game begins. Also, the probability of having n food pieces within FOV drops quickly as n increases and, for instance, for $n = 8$ it amounts to less than 0.5%. This further reduces the number of realistically possible FOV states by several orders of magnitude.

This suggests that it is difficult to build a sophisticated strategy based only on the current FOV state. Probably more may be gained by virtually extending the FOV, i.e., keeping track of board state as the ant moves. To enable this, we equip our ants with *memory*, which involves the following three components, each of them implemented as a two-dimensional array overlayed over the board:

- *food memory* F that keeps track of food locations observed by the ant during game,
- *certainty table* C that describes ant's belief in past board states,
- *track table* V that marks the cells visited by ant in the past.

At each move, the ant copies food locations from its FOV into F. Within FOV, old states of F are overridden by the new ones, while F cells outside the current FOV remain intact. As board states may change subject to opponent's behavior and make the memory state obsolete, we also simulate memory decay in certainty table C. Initially, the certainty for all cells is set to 0. During

the game, the certainty for the cells within FOV is always 1, while for cells (with food) outside FOV it exponentially decreases at each game move (with arbitrarily-chosen factor 0.9).

Table V stores ant's 'pheromone track', which is supposed to help avoid the already visited locations. It is initially filled with zeros, and visiting the cell is reflected by setting the corresponding table element to 1.

An ant may always escape the opponent's threat and avoid being killed. Therefore, we did not explicitly promote any chasing or avoidance strategies.

2 Ant representation

To evolve our ants, we used tree-based, strongly typed genetic programming (GP, [1]). The GP tree is expected to evaluate the utility of the move in particular direction: the more attractive the move, the greater tree's output. To benefit from rotational invariance, the same genetic code is invoked to evaluate multiple orientations. However, as ants are allowed to move in directions parallel to the coordinate axes *as well as* diagonally, we evolve two separate trees to handle these cases: a *'straight'* tree for handling main directions (N, E, S, W) and a '*diagonal*' tree to handle the diagonal directions (NE, NW, SE, SW)[1].

Thus, each individual is implemented as a pair of GP trees: 'straight' tree and 'diagonal' tree. We present the FOV state to the trees by appropriately rotating the coordinate system by a multiple of 90 degrees; this affects both FOV and the ant's memory. The orientation that maximizes trees' output determines the ant's move. Ties are resolved deterministically by preferring the earlier directions on the list.

Our GP trees involve operators that use three data types: *float* (F), *boolean* (B), and *area* (A). The area type represents a rectangular region, given as a quadruple of numbers: the coordinates of the rectangle center (relative to ant's current position, modulo board dimensions), and the dimensions of the rectangle. When mapping to the phenotype, the area dimensions are transformed in such a way that their sum effectively does not exceed 6. For instance, (2,3), (3,3), and (1,5) are correct area dimensions and are directly used in the phenotype, while dimensions (3,4) are mapped to (2,3) in phenotype.

The function set includes the following operators:

- Functions implementing terminal nodes:
  - Ephemeral random constants (ERCs):
    - Const(): Real-valued ($[-1; 1)$) ephemeral random constant for type F,
    - ConstInt(): Integer-valued (0..5) ephemeral random constant for type F,

[1] We considered using a single tree and mapping diagonal boards into straight ones; however, this leads to significant topological distortions which could possibly significantly deteriorate ant's perception.

        * Rect(): Ephemeral random constant for type A.
    - Functions based on ant's memory:
        * TimeLeft() – returns the number of moves remaining to the end of the game, i.e. 35−Time(),
        * Points() – returns the number of food pieces collected so far by the ant,
        * PointsLeft() – returns 15−Points().

- Functions implementing non-terminal nodes (operators):
    - Functions returning boolean (B) values:
        * IsFood(A) – returns *true* if area A contains at least one piece of food,
        * IsEnemy(A) – returns *true* if area contains the opponent,
        * Logic operators: And(B,B), Or(B,B), Not(B),
        * Arithmetic comparators: IsSmaller(F,F), IsEqual(F,F).
    - Functions returning float (F) values:
        * Scalar arithmetics: Add(F,F), Sub(F,F), Mul(F,F),
        * If(B,F,F) – evaluates and returns second child if first child returns true, otherwise evaluates and returns its third child,
        * Functions operating on the area:
            · NoFood(A) – returns the number of food pieces in area,
            · NoEmpty(A) – returns the number of empty cells in area,
            · NoVisited(A) – returns the number of cells already visited in the area,
            · FoodHope(A) – returns estimated number of food pieces that may be reached by the ant within two moves (assuming the first move is made straight ahead, and the next one in arbitrary direction).

Note that the functions that take the argument of area type compute their return value basing not only on FOV, but on the food memory table F and the certainty table C. For example, NoFood(a) returns the scalar product, constrained to area a, of table F (food pieces) and table C (certainty).

One should also emphasize that essentially all function used are unsophisticated. Even the most complex of them boil down to counting matrix elements in designated rectangular areas.

3 Evolutionary setup

The ants undergo competitive evaluation, i.e., they do not confront any externally provided selection pressure, but face each other. Moreover, our approach

does not involve explicit fitness measure. Rather than that, we combine the evaluation phase and selection phase, so that each tournament directly determines the outcome of selection. This feature makes our approach significantly different from most of contributions presented in literature. Apart from conceptual simplification, this also significantly lessens the computational burden.

To constrain the number of games needed for selection, we use single elimination tournament: the ants are paired at random, play matches against each other, and the winners pass to the next stage. The pairing and games repeat in consecutive stages. The winner of the last (final) match becomes the result of the selection process.

Trying to avoid the bias towards particular board state, we make each match consist of $2 \times k$ games. The games are played on k randomly generated boards. To provide for fair play, the contestants play two games on the same board, once as Ant 1 and once as Ant 2; we refer to such a pair of games *double-game*. To win the match, an ant has to win $k + 1$ or more games within the match. In the case of tie, the total score is taken into account. If there is still a tie, a randomly selected contestant wins.

To benefit from on-line batch compilation of individuals into C code (see Section 4), we used generational evolutionary algorithm. Typical evolutionary run lasted for 1500 generations, involved population of size 2000, and took 48 hours on a Core Duo 2.0 GHz PC (with two evaluating threads, each for one island). The mutation and crossover operators rely on default ECJ implementation, while the ERC mutation operators have been separately implemented for particular ERC nodes:

- For Const() we perturb the ERC with a random, normally distributed value with mean 0.0 and standard deviation 1/3.
- For ConstInt() we perturb the ERC with a random, uniformly distributed integer value from interval $[-1; 1]$.
- For Rect() we perturb each rectangle coordinate with a random, uniformly distributed integer value from interval $[-1; 1]$.

In all cases, we trim the perturbed values to the admissible intervals.

4 The technical implementation

We used ECJ (Evolutionary Computation in Java [2]) version 16 as the evolutionary engine for our experiments. However, to meet contest rules that require the ant code to be provided in C programming language, and to speed up the evaluation process, we developed a *game server* – an external program invoked by ECJ when needed. Technically, when ECJ comes to the evaluation phase, it serializes the entire population into one large text file, encoding each individual as a separate C function with a unique name. For each individual, this involves traversing its two GP trees and building a character strings that encodes the

Table 1: The settings of evolutionary run. Tournament size 5 implies that 6 games have to be played in the single elimination tournament.

Parameter	Value
Crossover probability	0.8
Mutation probability	0.1
ERC mutation probability	0.1
Individual initialization method	ramped half-and-half
Tree depth limit	8
Population size	2250
Generations	1350
No. games in a match	2×6
Tournament size	5

corresponding expressions in C, plus a preamble (a fixed piece of code containing function header and initialization of temporary variables).

The resulting file is then complied and linked with the game server. Next, the game server is launched and performs the entire evaluation and selection process, returning the ids of selected individuals as a result to the ECJ process. As all individuals are encoded in one C file, the compilation overhead is reasonably small, and it is paid off by the speedup provided by C (compared to Java). This entire process allows us also to monitor the actual size of C code, constrained by contest rules to 5kB.

The F, C, and V tables are implemented as static variables declared inside the C function.

5 How the BrilliANT evolved

To evolve our final contestant, we carried out a series of preliminary experiments that verified different variants of the approach, including:

- single population evolution vs. island model,
- explicit fitness (based on game score) vs. implicit fitness (selection based directly on tournament results),
- different variants of selection procedure.

The best evolved ant, called *BrilliANT* in the following, evolved in an experiment with evolutionary parameters that are summarized in Table 1. For the parameters not mentioned here explicitly, the ECJ's defaults have been used [2]. It is worth mentioning that many other evolutionary runs produced individuals that perform almost as well as BrilliANT.

The original encoding of BrilliANT's trees can be found in A.

5.1 How the BrilliANT was chosen

The process of choosing the best-of-run ant in case of 2-players game is computationally very demanding. We believe that the fairest and most exact method to choose the best ant is the round robin tournament with double-game matches; this is how BrilliANT was chosen. Let us, however, point out the numbers. Assuming that a match consists only of one double-game, the number of games needed is more than 10,000,000. For comparison, every generation of the evolutionary run involves only $4 \times 2 \times 2250 = 18000$ games. That means that this simple process of choosing the best-of-run ant is computationally as costly as 562 generations of evolution.

It is important to point out that the **BrilliANT was chosen in a completely autonomous way**. By that we mean that no human-made fixed opponent was used during this process.

6 Human competitiveness

In Ant Wars, one can consider two meanings of human competitiveness.

- **Direct competitiveness:** How does an evolved ant perform when playing a game with a *human*?
- **Indirect competitiveness:** How does an evolved ant perform when playing a game with a *program* devised by a human (called *HumANT* in the following)?

In the following, we consider both these cases.

6.1 Direct competitiveness: BrilliANT meets humans

We implemented a simulator that allows humans to play games against an ant (whether evolved or HumANT). One of us played 150 games against BrilliANT. The results are following:

player	games won	total score
Human	64	992
BrilliANT	86	1079

Even when we take into account the fact, that after playing 150 games in a row, a human player starts to make mistakes, the result (BrilliANT wins in more than 57% of games) can be definitelly considered as human competitive.

6.2 Indirect competitiveness: BrilliANT meets HumANTs

To assess the indirect competitiveness, we provided several manually designed ants (*HumANTs*). The process of designing HumANTs was iterative. Initially, we designed *GreedyANT*, however it was beaten by the ant evolved in on of the

first evolutionary experiments. Putting more thought in the designing process we created *SmartANT*. At this point we increased the maximum tree depth from 7 to 8 and increased the number of generations from 1000 to 2000. Using this setup, we were again able to evolved an ant that beats *SmartANT*. Having learned the lessons with GreedyANT and SmartANT, we finally designed *SuperANT* and *HyperANT*.

HyperAnt is the best ant we could develop manually. It tries to optimize the amount of food it can eat by considering 5 moves ahead. It is also equipped with a probabilistic memory model and a set of several end-game rules (e.g., *when you have 7 points, ignore the opponent and take the food*).

To our surprise, we could find an ant (*EvolANT*) that is as good as the HyperANT. When playing $2 \times 100,000$ games against the HyperANT, EvolANT wins 50,058% of them. This result is, however, statistically insignificant ($\alpha = 0.05$), thus we can only say that EvolANT and HyperANT seem to be equally good. Moreover, in order to find EvolANT we explicitly tested ants from the last generations against the *manually designed* HyperANT. As this, in our opinion, would violate the autonomy of an evolutionary process and introduce some degree of human intervention, we dropped HyperANT and decided to the submit fully autonomously evolved BrilliANT for the contest.

Quite interestingly, we observed that EvolANT was a bit overfitted – it heavily loses against BrilliANT (in 51.8% of $2 \times 100,000$ games). This is probably caused by the way it was chosen: by playing matches against HyperANT only. BrilliANT, however, loses against HyperANT (in 51.9% of $2 \times 100,000$ games).

The following table shows the results of Round Robin Tournament (2×5000 games in a match) between several mentioned ants.

player	matches won	games won	total score
EvolANT	6	34712	449858
HyperANT	5	34940	452247
BrilliANT	5	34472	457250
Evol2ANT	5	34178	458121
SuperANT	4	33934	450203
Noname1ANT	2	33439	450427
Noname2ANT	1	31524	437863
SmartANT	0	21669	365867

Names in bold represent the *evolved* ants. Apart from ants mentioned earlier, *Evol2ANT* is a 'younger brother' of EvolANT evolved in the same experiment, whereas *Noname1ANT* and *Noname2ANT* are some ants from very early, preliminary experiments.

Basing on the *games won* column, we can notice that the competition between top designed and evolved is very tight, and that the number of games per match must be large to spot the difference in players' performance.

7 Interesting behaviors observed in evolved ants

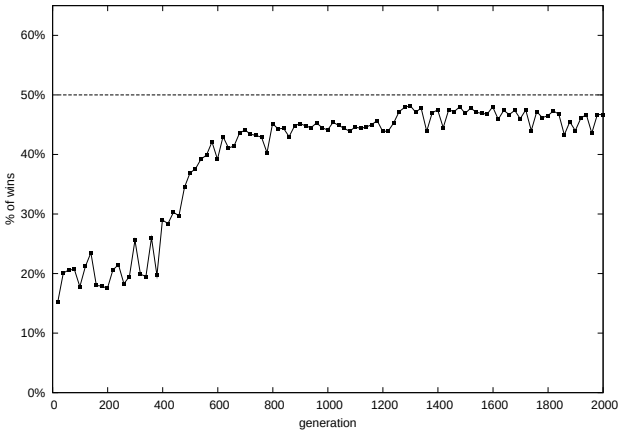
While experimenting with Ant Wars approach we observed several interesting phenomena. One of them was the emergence of **kamikaze ants.** The ants from several initial generations play poorly and are likely to be killed by the opponent. With time, they learn how to avoid the enemy and, usually at 200-300th generation, the best ants become perfect at escaping that threat (see Fig. 1b). However, in some scenarios such behavior may be undesired. In particular, a deadlock may occur where two ants hesitatingly walk around a single piece of food but refuse to consume it, fearing the opponent. When the game is coming to an end and the likelihood of finding more food becomes low, it may pay off to sacrifice ones life in exchange for food. In several experiments, we observed spontaneous emergence of such behavior at later evolution stages, after a long period of ruling of 'perfect escapers'. This effect can be shown in Fig. 1b.

Note that, although these figures have been obtained by playing matches between the best-of-generation ants and HumANTs, the evolutionary process as such is *completely autonomous* as it does not rely on external information but on competitive evaluation as defined in Section 3.

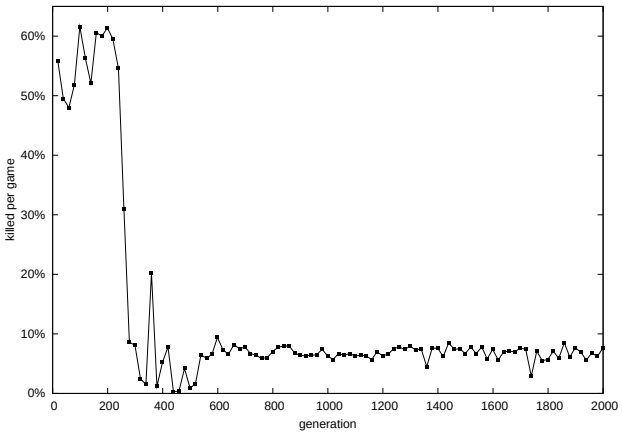
References

[1] J. Koza. *Genetic programming – 2.* MIT Press, Cambridge, MA, 1994.

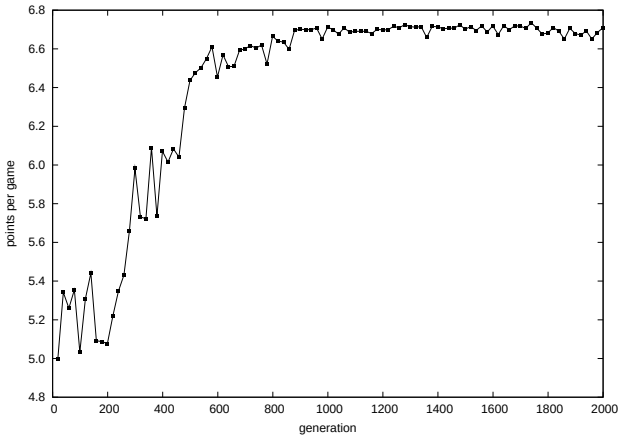
[2] S. Luke. ECJ evolutionary computation system, 2002. (http://cs.gmu.edu/eclab/projects/ecj/).



(a) Percent of wins against the hyperANT



(b) Percent of deaths per game



(c) Average score per game

Figure 1: Graphs show evolution dynamics for a typical process of evolution. Each point corresponds to an best-of-generation ant chosen on the basis of 500 games against the HyperANT. The presented values are averaged over 2×10000 games against HyperANT. It can be noticed that the evolution process usually converges around 1300 generation when the wining rate against a fixed opponent ceases to improve.

A The original encoding of BrilliANT's trees

Note that the C encoding uses different notation for the 'rect()' function: the four arguments correspond to lower left and upper right coordinates of the rectangular area relative to ant's current position (as opposed to midpoint and dimensions used in this original GP code).

The 'straight' tree:

```
add
    if
        not
            or
                isEnemy
                    rect(0,-3,2,1)
                and
                    isEnemy rect(-2,-1,3,2)
                    isEnemy rect(-1,0,1,2)
        if
            isFood
                rect(0,-2,1,2)
            mul
                foodHope
                if
                    isFood rect(-1,-3,1,1)
                    noFood rect(-2,-3,3,3)
                    add (add foodHope foodHope) (mul timeLeft foodHope)
            add
                sub
                    add (add foodHope foodHope) (mul timeLeft foodHope)
                    pointsLeft
                if
                    isEnemy rect(0,0,1,2)
                    sub (mul timeLeft foodHope) pointsLeft
                    add (add constint(5) pointsLeft) (noEmpty rect(-2,-2,3,2))
        add
            points
            if
                and
                    isEnemy rect(1,-1,2,1)
                    not (isFood rect(-1,-2,1,1))
                mul
                    add timeLeft timeLeft
                    timeLeft
            constint
                    2
    if
        and
            and
                isEnemy
                    rect(-1,-2,4,2)
                and
                    isEnemy rect(-2,-1,3,2)
                    not (isEnemy rect(-2,-1,3,2))
            and
                or
                    isEnemy rect(-1,0,1,2)
                    or (isEnemy rect(0,-1,2,1)) (isFood rect(-2,-3,1,1))
                isEnemy
                    rect(2,0,4,3)
        if
            and
                isEnemy
                    rect(0,-3,2,1)
                and
                    isEnemy rect(3,-2,4,2)
                    isFood rect(-1,0,1,1)
            if
                or
                    not (isEqual pointsLeft points)
                    or (isEnemy rect(0,-2,2,1)) (isEnemy rect(-2,0,2,4))
                if
                    and (isEnemy rect(2,0,2,1)) (isEnemy rect(-2,-1,3,2))
                    add foodHope (noEmpty rect(-2,-2,2,4))
                    add timeLeft (noEmpty rect(-2,-1,2,1))
                add
                    foodHope
                    foodHope
            if
                and
                    isFood rect(-1,-2,2,2)
```

```
                    not (isEnemy rect(-1,-1,2,1))
                if
                    isEnemy rect(-1,1,1,2)
                    mul timeLeft foodHope
                    add (add constint(5) pointsLeft) (noEmpty rect(-2,-2,2,2))
                pointsLeft
            if
                and
                    isFood
                        rect(0,-2,1,2)
                    or
                        isEnemy rect(0,-1,2,1)
                        or (isEnemy rect(0,-1,2,1)) (isFood rect(-1,-2,1,1))
                if
                    isEnemy
                        rect(0,-2,2,2)
                    points
                    if
                        isFood rect(0,-2,1,1)
                        add timeLeft (mul timeLeft foodHope)
                        add (add constint(5) pointsLeft) constint(3)
                const
                    -0.31385064
```

The ‘diagonal’ tree:

```
add
    if
        or
            or
                and
                    not (isEnemy rect(0,-1,2,1))
                    isEnemy rect(0,-1,2,1)
                isEnemy
                    rect(0,-1,2,1)
            or
                isFood
                    rect(-2,-2,5,4)
                and
                    isFood rect(0,-3,1,3)
                    and (isEnemy rect(-1,0,1,2)) (isEnemy rect(0,-3,2,1))
        if
            or
                isEnemy
                    rect(0,-2,2,2)
                or
                    isFood rect(0,-3,1,3)
                    isEnemy rect(-1,0,1,2)
            if
                and
                    isEnemy rect(-2,-1,3,2)
                    isEnemy rect(-1,0,1,2)
                mul
                    timeLeft
                    timeLeft
            constint
                3
        constint
            4
    const
        -0.31385064
    add
        if
            isEnemy
                rect(-2,-2,3,4)
            if
                not
                    and (isEnemy rect(3,-3,3,4)) (isEnemy rect(0,-1,2,1))
                noFood
                    rect(2,-4,5,4)
                if
                    isEnemy rect(0,-1,2,1)
                    add (add constint(5) pointsLeft) (noEmpty rect(0,-1,2,1))
                    pointsLeft
            if
                and
                    or (isEnemy rect(0,-1,5,1)) (isFood rect(-1,-2,4,2))
                    isEnemy rect(-2,-2,3,4)
                add
                    points
                    pointsLeft
                add
                    timeLeft
                    noEmpty rect(-2,-2,3,2)
        if
            and
```

```
    and
        or (isEnemy rect(-1,-1,2,1)) (isFood rect(-1,-3,1,1))
        isEnemy rect(-2,-2,3,4)
    or
        isEnemy rect(1,-1,2,1)
        and (isEnemy rect(0,-2,2,2)) (isEnemy rect(0,-1,2,1))
if
    and
        isEnemy rect(-1,-1,2,1)
        and (isEnemy rect(-2,-1,3,2)) (isFood rect(0,-2,1,1))
    noEmpty
        rect(-2,-2,3,2)
    constint
        3
if
    isEnemy
        rect(-2,-1,3,2)
    add
        add (noEmpty rect(-3,0,2,5)) (noEmpty rect(-3,-1,2,3))
        noFood rect(3,-3,1,4)
    add
        mul foodHope (mul timeLeft foodHope)
        add (add foodHope foodHope) (noFood rect(-2,-3,3,3))
```