# Genetic Algorithms in Autonomous Embedded Systems

University of Virginia Technical Report CS-2009-08
Chris Gregg

ECE 687: Embedded Systems – Spring 2009

## Abstract

*The performance and usefulness of autonomous embedded systems (AES) can be enhanced by providing them with artificial intelligence (AI). Because embedded systems are generally constrained by multiple factors (e.g., power consumption, processing speed, memory, etc.), fully-fledged AI implementations are not feasible for most AES designs. However, microprocessors targeted at embedded systems have improved to the point where it is possible to include certain AI methods in embedded designs. Genetic algorithms offer a modicum of AI that can successfully run on the newest generation of embedded processors, utilize minimal fixed storage, and are simple enough to integrate into an AES with beneficial results. This paper provides an argument for why genetic algorithms should be considered for autonomous embedded systems, and describes a method for implementing a genetic algorithm to control a small robot.*

## 1 Introduction

The nature of embedded computing necessitates an engineering design philosophy that places strong emphasis on size and power restrictions. Because of these constraints, embedded computers generally do not use the fastest and most powerful processors on the market, and therefore designs are limited in the *type* of processing they can do as a result. In other words, embedded computing relies on relatively simple algorithms, or depends on dedicated hardware to complete certain processing, such as digital signal processing, and encryption/decryption. Many Artificial Intelligence algorithms require intensive processing needs and large storage capacity, both of which seem at odds with an embedded system design. This is not to discount the fact that autonomous robots have been using rudimentary AI for many years, but there are powerful algorithms that have remained out of reach due to the aforementioned constraints. However, microprocessor vendors are improving their embedded processor offerings; it is common to see >2GHz, dual core processors touted for embedded use [1]. At this point in time it is possible to include more robust AI into embedded systems, and embedded system designers who think AI could be beneficial should consider certain options.

In particular, genetic algorithms can be successfully integrated into an embedded system design. Genetic algorithms are simple to construct, and do not necessitate a significant amount of storage, making them a sufficient choice for an embedded system with an adequate processor and small fixed storage. Furthermore, genetic algorithms are a good choice for embedded systems that gather data from sensors, as the algorithms can provide decision-making based on this data. Using GA for embedded systems (particularly, in robots) is not a new idea [7, 9, 11, 4], but paper authors frequently cite processing speed limitations, and conduct much of the research using simulations rather than real embedded computing. More recently, there have been advances in using GA for robotics [12], and processor speed has been less of an issue.

Genetic algorithms excel in solving problems (or, at least, tending towards a solution) that consist of solutions that entail a very large number of states. If an embedded computer is going to receive sample data from its sensors and make a control decision on that data, it might be impossible to gather data in enough configurations to make a formulaic decision about how to continue. In this case, a GA could be used on the sample data to get a solution that tends towards the optimal solution. For example, many embedded systems run a real-time schedule. This schedule is usually determined when the design is being created, and it is fixed thereafter. What if, however, the embedded computer needed to change the schedule based upon data from its sensors? Optimal scheduling for a complex real-time system is a nonlinear problem that cannot be solved easily[10]. Yet, a GA could serve to find a decent solution in a limited amount of time. For instance, assume a designer is building an outdoor embedded system that needs to run its schedule depending on weather conditions. The system could take some data about the current conditions, and then run a GA for a certain number of iterations to determine a workable solution without resorting to trying to solve a hard problem.

Section 1 gives a background primer on genetic algorithms. Sections 3 discuss resource requirements that are necessary for GA use in a typical embedded system, and how various embedded systems could benefit from using GAs. Section 4 describes the specific use of a GA to control a robot, section 5 describes the limitations that come with using GAs in embedded systems, and finally, section 6 concludes the paper and provides suggestions for future work in the area.

## 2 Background

Genetic algorithms are a set of problem solving strategies that are modeled on the evolutionary theory of survival of the fittest. GAs were initially used (in the 1950s) to study actual biological evolution, and later the ideas were expanded to use GAs for general ma-

[1] See, for instance, http://news.thomasnet.com/fullstory/556779

| Biological Evolution | Genetic Algorithms |
|---|---|
| 1. Individual organism has chromosomes that encode information about the individual | 1. Individual chromosomes are created, encoding the information about the problem to be solved |
| 2. Organisms mate, with offspring carrying a combination of the parents' chromosomes | 2. Chromosomes are recombined with one another, depending on individual fitness level (i.e., fitter chromosomes are more likely to solve the problem, and are thus more likely to be recombined) |
| 3. Some organisms have mutations in their chromosomes, which may or may not enhance their ability to survive | 3. Some chromosomes are chosen for mutation, in which random bits are changed |
| 4. A percentage of organisms that are less likely to survive will die off, and a lower percentage of those more likely to survive will die off. | 4. The chromosomes with the highest fitness level are kept, while those with the lowest fitness level are discarded |
| 5. Repeat from step 2 | 5. A decision is made whether to terminate, if a predetermined fitness level is reached, or if a fixed number of iterations have been completed. |
| | 6. If not terminated, the process repeats from step 2 |

Figure 1: Biological Evolution -vs- Genetic Algorithms

chine learning and problem solving. Today, GAs are used in many scientific research fields (as well as computer science) to solve design problems (e.g., microchip design, antenna design), scheduling, game-theory[3] etc., and they are also used in the commercial sector for uses as diverse as stock market prediction[8] to data mining[2].

Because genetic algorithms are based on a biological model, much of the terminology is taken from the biological analogy. Figure 1 shows the analogy in detail. The basic building block in a GA is the *chromosome*, which is generally a binary bit string that holds information about the problem to be solved. The algorithm initializes a large number of initial chromosomes to random bit strings (although they can be seeded with real data if the information is available), and then the *fitness* of each individual chromosome is evaluated. The fitness is based on how well the chromosome solves the problem, and determining the *fitness function* can be challenging, but does not happen in real-time; it is programmed into the algorithm. Once the fitness level of the individual chromosomes are calculated, the algorithm then chooses a fixed number of chromosomes in a random manner whereby chromosomes with a higher fitness level are weighted more heavily and are more likely to be chosen. The chosen chromosomes are then either *recombined* with other chosen chromosomes, or they go through a random *mutation* where some of their bits are randomly changed. Both recombination and mutation could occur for any chromosome. In the recombination process, two chromosomes are combined together such that their *children* (generally) have half of each *parent's* bits, analogous to biological reproduction. There are numerous methods for recombination, but a simple one entails picking a random point in a chromosome's bit data, and swapping the leading or trailing bits with the other parent's respective leading or trailing bits.

In the mutation process, an arbitrary number of bits from a chromosome are changed, dependent on a random variable associated with each bit. The reason for mutation is two-fold: first, mutation reduces the chance that the population converges upon a solution that is not optimum (a local minimum); and second, the diversity of the population is increased and the introduced mutation may start the population heading towards the solution.

Once the recombination and mutation phase is complete, the fitness of the complete population is reevaluated and re-ranked, and total number of chromosomes is returned to its original number. There are a number of different ways to pick the selection of the chromosomes that will go on to the next iteration; for example, the chromosomes that show the lowest fitness (either original members or children) can be removed (called the *elitist* method), or another round of fitness-proportional selection can occur, similar to how the chromosomes were chosen for recombination (called *Roulette-wheel* selection). After the selection is complete, the algorithm evaluates whether or not the process should continue, based on either a number of complete iterations, or whether a pre-determined level of fitness has been reached by the top chromosome. If the termination criteria have not been reached, the algorithm continues again with the recombination selection and the updated batch of chromosomes.

It should be noted that this algorithm is not guaranteed to produce a solution, but the law of large numbers predicts that it will tend towards a solution, as long as the fitness function was chosen correctly.

# 3 Resources Required for Genetic Algorithms in Embedded Systems

Deciding whether to run a genetic algorithm on an embedded system must include an analysis of the minimum resources necessary for the system to work. The following subsections describe the main resources a designer must consider.

## 3.1 Processor and Operating System

As mentioned in the introduction, genetic algorithms can be processor-intensive. Therefore, an embedded system that utilizes GAs must have a processor that is up to the task, and have an operating system capable of running the program the algorithm is coded in. If the embedded system is going to be run from a microprocessor, there are many options that will suffice, and the GA would probably run alongside (or as a function-call) in the operating system that will be used. If, on the other hand, the embedded system is designed to run on a less powerful microcontroller, then the options change. Some microprocessors could probably handle a rudimentary GA, but they would struggle with the task. In that case, it would probably be best to upgrade the entire embedded system to a fast microprocessor, or include a microprocessor as well as the microcontroller. This is

the proposed solution in section 4, where two microprocessors are used to control the robot.

Genetic algorithms are simple enough that an operating system only needs to provide a couple of services in order for a GA to run. There must be a random or pseudo-random number generator available to the GA routine, and there must be the ability to process integer bitstreams, preferably at the bit level. Preferably, the operating system should be programmed in a high level language (e.g., C, Java, etc.), but there is nothing special about a GA that it cannot be programmed in assembly code or machine language if necessary.

The other solution would be to use a hardware-implementation of a GA, and examples of such exist already in the literature. Aporntewan and Chongstilivatana describe an FPGA implementation of "The Compact Genetic Algorithm," described by Harik, et. al.[5, 1]. This type of solution would be ideal for a situation where a GA is desired but there are other concerns such as memory and power considerations.

### 3.2 Memory

The amount of memory a GA needs is dependent on the parameters of the fitness function, on how many chromosomes the designer wants to support, and on the bit length of the chromosomes. The greater the number of chromosomes, the larger the memory requirement. Because GAs generate the initial group of chromosomes randomly, SRAM or DRAM is more important than flash or other nonvolatile memory. The Compact Genetic Algorithm described in section 3.1[5] fits nicely in a limited amount of memory.

### 3.3 Power

Power consideration is also dependent on the nature of the GA that will be on the embedded system. In this case, the primary parameters are both the number of chromosomes and (more importantly), the number of iterations that the designer wants to run. The greater the number of iterations, the greater the power drawn. If power is a concern, it would be wise to design the GA to stop after a certain number of iterations rather than when an acceptable fitness level is reached, because the latter case could involve much longer processing time, and it would be difficult or impossible to predict how much power would be necessary in that case. Regardless, the designer must consider the trade off between power conservation and precision of the GA, as a shorter number of iterations will produce a result with less precision.

## 4 An Example of a Genetic Algorithm in an Embedded System

Throughout this paper it has been mentioned that the field of robotics can benefit from using genetic algorithms in real time. A classic problem in the field of robotics is the problem of obstacle avoidance. The problem is stated as follows: *A ground-based, usually wheeled*[2] *robot needs to get from point* A *to point* B*, with one or more obstacles in its path*. The robot has a set of sensors that can detect the obstacles (e.g., by bumping into the obstacles, or by using sonar or visual imaging), but the goal is to reach point $B$ in a reasonable amount of time without straying too far from the best path. Numerous algorithms and solutions appear in robotics literature, but the following example will show how apply a genetic algorithm to the task.

This example uses a *Roomba Scheduler* autonomous vacuum robot (shown on the left in Figure 2) made by iRobot. iRobot enables all Roomba vacuum cleaners with a programmable serial interface, and Roombas make an excellent testbed for basic robotics research[13]. Various third party manufacturers sell Bluetooth adapters that allow a radio-controlled interface, and this example uses the Firefly Bluetooth adapter made by SparkFun Electronics. For this project, the robot is controlled by a laptop computer running the Apple OS X operating system, but the control software is written in Java, and the next stage of the project is to port the software to a mobile phone running the Windows Mobile operating system, and it can then be physically attached to the robot for fully autonomous behavior. At that stage, the robot/mobile phone combination would fit a common definition of an embedded system.

The *Roomba* is equipped with numerous sensors (e.g., a collision detection bumper, cliff-avoidance, infrared) , and for this experiment the bumpers were used to detect when an obstacle collision occurred. The robot API includes functions for detecting when collisions occur, for driving the robot forwards, backwards, and in a circular motion, and rotating the robot along its center axis. It also has the ability to report on driving distance, and has a range of speeds.

The strategy for solving the problem listed above is as follows:

1. Start by positioning the robot at point $A$, pointed in the direction of point $B$. The distance from point $A$ to point $B$ is then programmed into the robot, and the program is started. The robot starts driving towards point $B$.
2. When the robot hit the first obstacle, it uses a genetic algorithm to determine how to proceed. This algorithm is detailed below. Once a decision has been made upon which direction to proceed, the robot continues in that direction.
3. If the robot is not facing point $B$ during its next forward progression, it goes a maximum of 256cm in a straight line before turning back towards point $B$, and the robot continues forward.
4. The process is then repeated from step 2.

The genetic algorithm from step 2. above has the following variables that contribute to a 16-bit chromosome:

**distance:** the distance traveled in a straight line since the last collision (8-bits, 256cm is maximum)

[2] Bipedal robots have been hot topics for research in recent years; Honda's ASIMO is a fascinating example*[6]*

**sideHit:** the side on which a collision has occurred. (2-bits, 00 = 11 = no hit; 01 = hit on right side of bumper, 10 = hit on left side of bumper)

**goalDirection:** the direction of the goal relative to the direction the robot is currently heading (6-bits, with the maximum value of 111111 straight ahead, and 000000 directly behind. Odd numbers denote the angle left of the target, and even numbers denote the angle to the right of the target)

Using this 16-bit chromosome, a fitness function is defined as follows, with the $bumpFitness$ variable used as a temporary variable to give proper weight to the bump. $bumpFitness$ is set to 0 if a collision happened on the side of the robot that points generally in the direction of point *B*, or if there was no collision but the robot is not headed towards point *B*. $bumpFitness$ is set to 5 if a collision happened on the opposite side to the direction of point *B* (a good thing), and it is set to 10 if there was no collision and the robot is headed directly towards point *B* (a very good thing). The calculation for the fitness level is the following formula:

$fitnessLevel = distanceTraveled + bumpFitness + goalDirection$

A chromosome with a higher fitness level leads to a greater probability for being chosen to recombine, and the chromosome with the highest fitness level will be chosen to provide the direction for the follow-on leg. As noted earlier in the paper, choosing a proper fitness function is a difficult task. Determining the proper weights for each parameter usually amounts to testing the system with many different possibilities. In this case, $fitnessLevel$ was chosen for simplicity for the example, and with a little intuition as well.

The algorithm for the genetic algorithm is listed in Algorithm 1. The idea can be summarized as follows: once a collision has occurred, the robot should assess its surroundings using sensors. This is accomplished by performing ten tests in random directions (but not backwards). These tests will provide the "seed" chromosomes to augment a large number of randomly chosen chromosomes. Using this data, the genetic algorithm will make a choice about which direction to travel based upon the fitness of the seed chromosomes recombined and mutated with the random chromosomes. For example, if after a collision, one of the directions the robot tries results in 256cm of travel in a direction that is relatively straight towards point *B*, this chromosome will be a good contender for the highest fitness level. However, because of the recombination process, there will also be many other contenders that will also vie for the most fit. Because of the small number of "real" tests, it may indeed be better for the robot to choose a direction that is not exactly in the direction of the most fit "real" case, and this is the strength of a genetic algorithm: the choice is based on an evolutionary model, which should tend towards a decent solution.

Once again, choices had to be made for the parameters of the genetic algorithm. Ten thousand random chromosomes were chosen because there needs to be sufficient diversity in the population in order to have a wide range of fitness levels. While it might seem that having only 0.1% tested chromosomes in the group would lead to the tested cases being overwhelmed, remember that they have a higher probability of having a decent fitness level simply because they were actually tested. Furthermore, if some random chromosomes do indeed produce better fitness levels, by definition they should be chosen, even if they don't immediately lead to the most obvious solution. In addition, the recombination of chromosomes with the random chromosomes will tend to produce a healthy set of high-fitness chromosomes. Finally, consider the following situation: if the robot enters a set of obstacles that surround the robot such that all of the test cases lead to almost instantaneous collisions, the best fitness chromosome will undoubtedly send the robot back in the direction from which it came, which is exactly what should happen.

**Algorithm 1** The Genetic Algorithm for Robot Obstacle Avoidance

1. Choose 10,000 random chromosomes
2. Choose 10 additional "seed" chromosomes, using the following method:
   (a) Have the robot travel in reverse to the last point at which it was turned, or 256cm, whichever is less.
   (b) Select a random angle, between -90 degrees and +90 degrees, and spin the robot on its center axis in the chosen direction.
   (c) Have the robot proceed forward, until it either hits an obstacle, or it travels 256cm.
   (d) Return the robot to its previous position for the remaining 10 seeds.
3. Randomly choose 50% of the chromosomes to go through recombination.
4. Randomly choose 10% of the chromosomes to have five bit-mutations each.
5. Sort all of the chromosomes by fitness level, and remove all but the top 10,000.
6. Use the chromosome with the top fitness level to determine the direction that the robot should travel.

Once the algorithm has completed and returned the chromosome with the highest fitness, the robot continues down the path dictated by the *goalDirection* variable embedded in the chromosome. If the robot then travels 256cm before hitting another barrier, it will then turn to face point *B* and head back in that direction. This keeps the robot moving in the general direction if no barriers are hit (thus not triggering the genetic algorithm).

Other considerations that are included in the code (and are independent of the genetic algorithm itself) are related to keeping track of where point *B* is in relation to where the robot is. Each time the robot makes a turn, the relative direction of point *B* is computed and stored.

Initial tests on the algorithm have been promising. Despite the relative simplicity of the algorithm and associated coding, there are still some bugs in the code that need to be worked out. However, it is intriguing to watch the robot as the algorithm progresses, and to watch

Figure 2: The *Roomba Scheduler* robot and T-Mobile SDA mobile phone with the Windows Mobile 5 operating system. The port cover for serial communications can be seen in the lower right side of the robot. Figure copyright iRobot, inc., and T-Mobile, inc.

it sometimes make a choice that doesn't make sense. Just as often, however, it makes a choice that is spot-on, or it makes a series of choices that eventually lead to extricating itself from a set of barriers and back into the correct direction.

Unfortunately, actually getting to point $B$ is somewhat difficult for the robot, for a couple of reasons. First, the distance calculations do not seem to be as precise as the iRobot API reference would lead one to believe. The API states that it reports "The distance that Roomba has traveled in millimeters since the distance it was last requested," but the tests show that this can vary by up to 10-15% of the actual distance traveled, and this imprecision can lead to disastrous results when trying to keep track of point $B$. However, the algorithm can be seen to be working, and with better distance precision, the results would be more favorable. Second, there are barrier placement positions that will lead to a very long period of test/recompute cycles. Again, this is where judicious parameter choices can be made to mitigate this sort of occurrence.

Figure 3 shows how the robot might travel between two points with obstacles. Because of the random nature of the GA, this is one of an infinite number of paths, however many paths will look somewhat similar. One takeaway should be that the genetic algorithm is only doing part of the work (albeit a significant part), and other correction mechanisms (e.g., turning towards target after 256cm of obstacle-free travel) are at work as well.

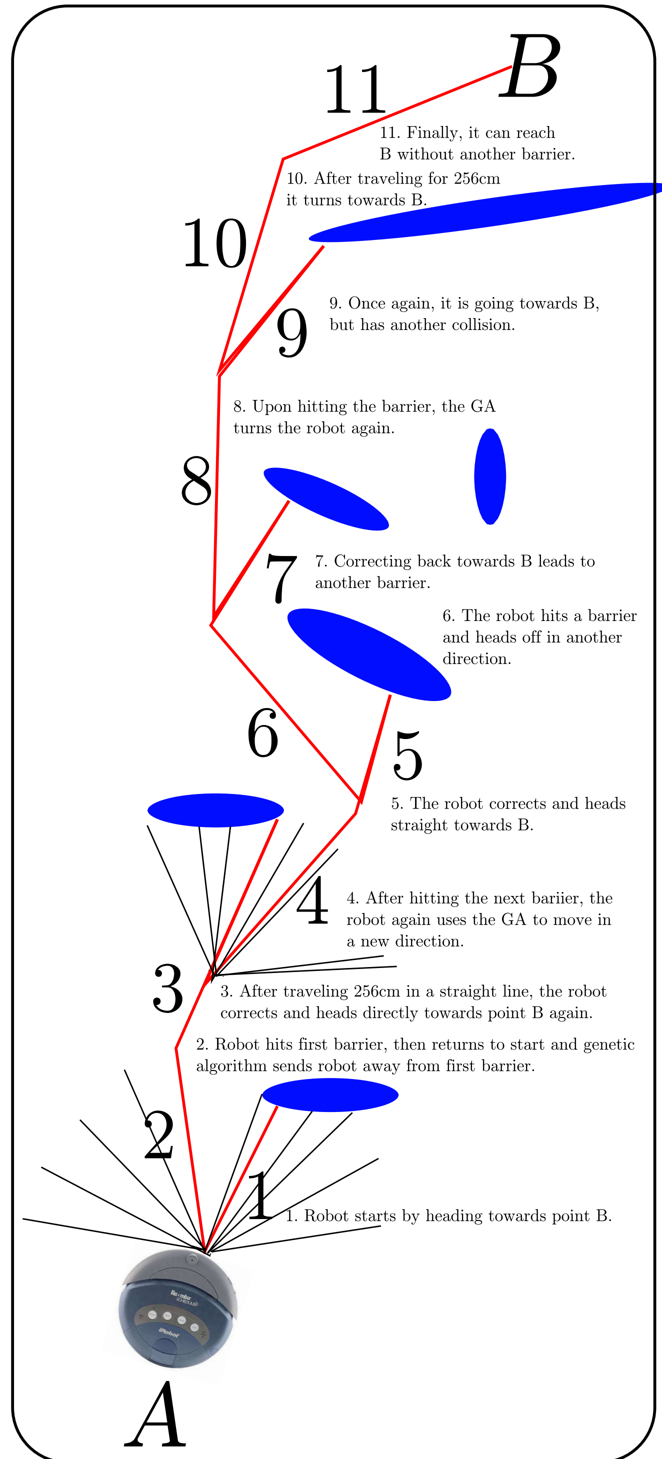


Figure 3: Example Path for Algorithm. The test movements are shown as darker lines for steps (2) and (4).

## 5 Limitations

This paper has shown that it is possible to implement genetic algorithms in an embedded system. However, there are some caveats that should be considered before an embedded system designer decides to use a GA in a project. Some of these stipulations are listed below:

**Resource Concerns:** As described in section 3, the designer has to ensure that the cost of implementing the necessary resources to

use GAs in a project is worthwhile. If there are other solutions that need less processing ability, need less memory, or consume less power, those options should be investigated.

**Time Constraints:** Depending on the parameters of the GA code, GAs can take time to run. GAs can work if there are real-time constraints, but the designer needs to be careful to allot enough time for the algorithm to run to completion, and should certainly constrain the number of iterations to terminate the algorithm, rather than have it run until a fitness level is reached.

**Inefficiencies:** A designer should not use a genetic algorithm just for the sake of using one. Many problems can be solved efficiently with a deterministic algorithm, and if such an algorithm exists, it will undoubtedly be more efficient than an equivalent GA. While it is true that GAs are relatively simple to implement, in many cases it is worthwhile to spend the extra time devising a more complex algorithm that does not have the shortcomings of a GA. That said, sometimes it is much more efficient to implement a well-targeted GA that can get a decent answer simply.

**Determinism:** It goes without saying that if a design needs a deterministic solution to a problem, it is probably best to forgo using a GA. There are cases where letting a GA run until reaching a solution is acceptable, but there are numerous caveats to this process, and it is hard to think of an embedded system example where this would be the case.

# 6 Conclusion and Future Work

Embedded systems will always incur more limits than their equivalent general purpose computing cousins. Some design ideas will always need too many resources that are not easily available to an embedded system, and embedded system designers will continue to have to make trade offs, or to look for alternate solutions. Until recently, genetic algorithms fell into this category, because they require a relatively high level of processing ability, and this was not feasible for slow, low power processors. At this point in time, however, processors designed for embedded systems have become faster, and memory has become cheaper and more abundant. It is therefore possible to successfully utilize GAs in embedded systems, and if a designer sees an opportunity to use a GA, she should recognize the limitations, but consider it nonetheless. GAs are powerful and flexible, and excellent solutions to many problems that embedded systems designers face.

Genetic algorithms will always benefit from more processing power, and as embedded processors increase in speed, it will continue to get easier to utilize GAs. Furthermore, GAs perform particularly well in parallel processing environments, so as multiple cores continue to, and general purpose graphical processing units (GPGPUs) start to arrive in the embedded computing arena, it will become even easier to use GAs for embedded designs. Continued work in hardware implementations are also feasible, although software implementations will probably be easier for the majority of designers.

Finally, genetic algorithms comprise just one part of the artificial intelligence field, and embedded systems (particularly robotics) will continue to benefit from faster, more powerful embedded processors. Future work should look towards finding out the best way to implement AI into embedded systems, whether it is with genetic algorithms or any of the host of other AI methods.

# References

[1] C. Aporntewan and P. Chongstitvatana. A hardware implementation of the compact genetic algorithm. volume 1, pages 624–629 vol. 1, 2001.

[2] Wai-Ho Au, K.C.C. Chan, and Xin Yao. A novel evolutionary data mining algorithm with applications to churn prediction. *Evolutionary Computation, IEEE Transactions on*, 7(6):532–545, Dec. 2003.

[3] K. Chellapilla and D.B. Fogel. Anaconda defeats hoyle 6-0: a case study competing an evolved checkers program against commercially available software. volume 2, pages 857–863 vol.2, 2000.

[4] D. Floreano and F. Mondada. Evolution of homing navigation in a real mobile robot. *Systems, Man, and Cybernetics, Part B: Cybernetics, IEEE Transactions on*, 26(3):396–407, Jun 1996.

[5] GR Harik, FG Lobo, and DE Goldberg. The compact genetic algorithm. *IEEE Transactions on Evolutionary Computation*, 3(4):287–297, 1999.

[6] M. Hirose and K. Ogawa. Honda humanoid robots development. *Royal Society of London Philosophical Transactions Series A*, 365:11–19, January 2007.

[7] K. Kojima and K. Ito. Autonomous learning algorithm and associative memory for intelligent robots. volume 1, pages 505–510 vol.1, 2001.

[8] R. J. Kuo, C. H. Chen, and Y. C. Hwang. An intelligent stock trading decision support system through integration of genetic algorithm based fuzzy neural network and artificial neural network. *Fuzzy Sets and Systems*, 118(1):21 – 45, 2001.

[9] Wei-Po Lee, John Hallam, and Henrik Lund. Learning complex robot behaviours by evolutionary computing with task decomposition. *Learning Robots*, pages 155–172, 1998///.

[10] D. Montana, M. Brinn, S. Moore, and G. Bidwell. Genetic algorithms for complex, real-time scheduling. In *IEEE International Conference on Systems, Man, and Cybernetics*, volume 3, pages 2213–2218. Institute of Electrical Engineers Inc (IEEE), 1998.

[11] Yogo Takada, Toshiaki Tamachi, Satoshi Taninaka, Toshinaga Ishii, Kazuaki Ebita, and Tomoyuki Wakisaka. Development

of fish robots powered by fuel cells: Improvement of swimming ability by a genetic algorithm and flow analysis by computational fluid dynamics. *Bio-mechanisms of Swimming and Flying*, pages 233–245, 2008///.

[12] F.L. Tong and M.Q.H. Meng. Genetic Algorithm Based Visual Localization for a Robot Pet in Home Healthcare System. *International Journal of Information Acquisition*, 4(2):141–160, 2007.

[13] B. Tribelhorn and Z. Dodds. Evaluating the Roomba: A low-cost, ubiquitous platform for robotics research and education. In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA'07)*, pages 1393–1399, 2007.