# Using a Genetic Algorithm to Evolve a D* Search Heuristic

**Andrew Giese and Jennifer Seitzer**
**University of Dayton**
gieseanw@gmail.com, seitzer@udayton.edu

**Abstract**

Evolutionary computation (EC) is the sub-discipline of artificial intelligence that iteratively derives solutions using techniques from genetics. In this work, we present a genetic algorithm that evolves a heuristic static evaluation function (SEF) function to be used in a real-time search navigation scheme of an autonomous agent. This coupling of algorithmic techniques (GAs with real time search by autonomous agents) makes for interesting formalistic and implementation challenges. Genetic evolution implies the need for a fitness function to guide a convergence in the solution being created. Thus, as part of this work, we present a fitness function that dictates the efficacy of a generated static evaluation function. In this work, we present algorithmic and formalistic designs, implementation details, and performance results of this multi-layered software endeavor.

## Introduction

The A* Algorithm is a greedy best-first search algorithm that is used to calculate an optimal path between two points, or nodes, on a graph (Hart et. al 1968) The algorithm can be adapted to a run in real-time by way of restarting execution as new environment information becomes available, called a Dynamic A* (D*) search. The A* search uses a static evaluation function (SEF) that uses heuristics to find a path of state transformations from a start state to a goal state. The SEF assesses the merit of each child state as it is generated by assigning it a numeric value based on information about that state. The score allows the A* to direct its search by prioritizing the expansion of child nodes that could potentially expand into a goal state while neglecting child nodes that are less likely to lead to a goal.

In a real time environment, information about the actual goal state is unavailable and unobtainable for any given iteration of the search for an agent (because it is out of range of the agent's sensors). Therefore, the SEF must direct the search to the most appropriate state that anticipates system information as it becomes available. In our work, the SEF seeks to maximize some aspects of an agent's state while minimizing others. By evolving a weight on each "aspect-variable", we are able to create offline a highly effective SEF that can predict obstacles and challenges that occur in real time during the execution of D*. In this paper we present the offline pursuit of using a genetic algorithm as a mechanism to evolve an optimal SEF to be used in the real-time execution of A*. Additionally, we use the simulator that will eventually benefit from this optimized SEF to provide feedback in the evolution process. That is, the simulator serves as the fitness function for the evolving SEF.

This work is novel in that it combines techniques of evolutionary computation using genetic algorithms and the use and refinement of a heuristic for the D* algorithm. There are many applications of genetic algorithms in diverse domains such as bioinformatics (Hill 2005), gaming (Lucas 2006), music composition (Weale 2004), and circuit optimization (Zhang 2006). Additionally, work in D* has been studied and developed in theory (Ramalingam 1996) as well as specific applications such as robotics (Koenig 2005). We are using the all-inclusive examination that genetic algorithms affords us to find the perfect (or near perfect) heuristic function for a derivative of the very traditional AI search, A*.

## The A* and D* Algorithms

In this work, the evolution of a static evaluation function using a genetic algorithm is applied to an autonomous agent operating in an environment provided by Infinite Mario Bros., an open-source, faithful recreation of Nintendo's Super Mario World. The agent (Mario) uses a realtime execution of an A* search, called D*, to direct its movement through the environment to ultimately reach the goal (the end of the level). Mario may use information about what is currently visible onscreen, but beyond that nothing is known, making a calculation of an actual path to the goal impossible. Therefore, the SEF of the D* must direct Mario towards states that are on the path to the goal.

Mario has a total of thirteen distinct action combinations that allow him to negotiate the environment. These are move left, move right, duck, and two others—jump and speed—that can be used in combination with the other actions and each other. Jump allows movement along the y-axis, and can be used in combination with right, left, and duck. Speed allows for faster movement left or right, and higher jumps. This means that from any state, there could be up to thirteen child nodes. Since the agent must operate at 24 frames per second, the agent is allotted approximately 40 milliseconds to perceive its current state, decide what to do next, and return a chosen action. With up to thirteen child nodes from any node in the search tree, any algorithm that decides what Mario is going to do next must do so quickly and efficiently. A "brute force" approach that analyzes all possible children was infeasible given

available computing machinery, and therefore a dynamic D* search is more appropriate.

The SEF of a D* search uses information about a state to direct the search efficiently. For Mario, much information is immediately available from each percept provided by the environment. This information includes the position of Mario, the amount of damage Mario has taken, the positions and types of enemies onscreen, and position and types of obstacles onscreen. Other information can be tracked over time, like number of kills, X velocity, Y velocity, coins collected, time remaining, etc. The task of our system was to discover what effects, if any, the values of these variables should have on the valuation performed by the SEF of a node in the search graph. A high value for a variable might proportionally increase the cost to transition to that state, or conversely could proportionally decrease the transition cost.

## The System

In 2009 and 2010 Julian Togelius of the ICE-GIC held a competition for entrants to create an autonomous agent (bot) that would play Markus Persson's Infinite Mario Bros. the best. "Best" in this sense means the distance a bot could travel within a given level and time limit. If two bots finished a level they were awarded equal scores, but if neither finished, the bot that travelled furthest was deemed better. In both iterations of the competition, the same bot was victorious. This bot was written by Robin Baumgarten (Baumgarten). Robin's bot used a D* search coupled with an accurate method for expanding child nodes, and a human-generated static evaluation function for the D*. Our system is a heavily modified version of Robin's, with the majority of the A* rewritten for legibility and efficiency while the means to produce child nodes was mostly preserved.

Every 24 frames, the environment provides the agent with a percept that includes the locations and types of all sprites on the screen, including Mario. The agent must return an action to the environment that the environment then effects upon Mario. For each percept received, the agent runs an A* search for 39ms or until the agent has planned out to 15 levels of the search tree. The agent keeps an internal representation of the world, and tracks Mario's x and y velocities among other things not provided by each percept.

After ensuring that its internal representation is consistent with the environment-provided one, the agent begins an A* search from Mario's current position and velocity. Children are generated by considering which actions are available to Mario at any node. That is, a child state reflects where Mario would be and how fast he would be moving if performed action A from node M. (Figure 1) A child state also informs the search of whether Mario would take damage, die, kill an enemy, collect a coin, etc. upon performing action A from node M. A static evaluation function provides weights on Mario's X position, X velocity, Y position, Y velocity, Mario's damage taken, whether Mario is carrying a shell, Mario's X position multiplied by X velocity, and Mario's Y position multiplied by Y velocity. These weights are values between -1 and 1. After multiplying weights to their associated state variables, the sum of products forms the final SEF score for that node.

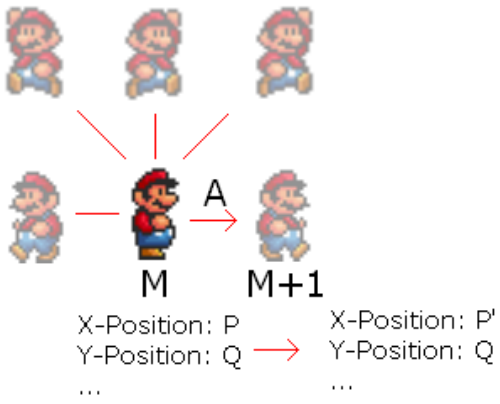


Figure 1

This SEF score is an estimation of the amount of work required to reach a goal state from the current node, and as such nodes with lower SEF scores are preferable. As an A* algorithm dictates, the level of the search tree at which the node was discovered is also added to the score. This is the "greedy" part of an A* search where not just a solution is desired, but the best solution. For Mario, the cost to transition from one state to another is uniform; all neighboring states have the same arc cost to travel to a neighbor. Adding the sum of arc costs into the SEF score for a node is a means by which the "work" required to reach a node in the graph is represented, so that if the same node is reached by two separate paths, the shortest path is favored. Since the D* search operates in a partially observable world, an admissible heuristic is difficult to discern, hence the motivation for a Genetic Algorithm to search for the optimal weights to apply in the SEF.

During the D* search, children nodes are generated from the current state of the agent. Generated children are placed on an open list sorted from lowest to highest scores. The child with the lowest score is taken from that list, and its children are generated. This process repeats until the agent has searched for 39ms or has searched 14 states (empirical number), at which point it returns the action that leads to the most optimal path for the current available information.

The values for the weights used in the agent's SEF mentioned above are deemed to be "unknown" to the system, and are provided via parameters supplied by an external entity, in this case a Genetic Algorithm. The

genetic algorithm is implemented as defined in (Russell and Norvig 2003). The chromosome being evolved is an array of 8 floating point values, each between -1.0 and 1.0. The mutation rate was 1%.

Each generation of chromosomes was tested for fitness by running a simulation on a training level where the agent used the chromosome's genes as the weights on state-variables evaluated by the SEF in a D* search. The fitness of the chromosome was a summation of Mario's distance travelled, and if he completed the level, also the remaining time Mario had to complete the level. A higher fitness score indicates a better, or more fit, chromosome. This is in contrast to the Static Evaluation function where a lower score indicates a more ideal state.

The test level that each bot was scored on had a variety of characteristics. The most important of these is that the level was short. As each chromosome needed to be used in an actual bot, a single fitness test could last upwards of a minute even if the bot could finish the level successfully. A short level guaranteed that if a bot was going to finish a level, it could do so without much time spent. The second characteristic of the level was an imposed time limit. This time limit places an upper bound on the possible time a bot could spend in a level. Slow bots, bots that stood still, or bots that got stuck therefore all required a maximum of N seconds to evaluate.

An ideal level must also contain challenges and obstacles that a full level will have on its maximum difficulty. These challenges include portions with a high volume of enemies, some which that cannot be destroyed by landing on their heads; portions with Bullet Bill towers of varying heights; gaps of varying width; pipes with Piranha Plants leaping out of them; and portions with mixtures of these scenarios.

### Optimization to D* and GA

The D* search still performed sub-optimally given computing hardware, so the search tree needed to be pared down. Paring the tree followed a simple formula: if two child nodes generated the same score from the SEF, the first child node was kept and the other discarded. In a further endeavor to pare the tree, the maximum degree for a node was reduced from 13 to 11 by discounting nodes reachable through the action of ducking by the Agent. In an effort to avoid a bias in the reproduction phase of the genetic algorithm, a generated and tested chromosome was only added to the population if either its fitness score was unique or, failing that, the genes on the chromosome were unique among the chromosomes with the same fitness. If this precaution was not taken, a glut of identical chromosomes with the same score could skew the parental selection process unfairly.

## The Experiment

The Experiment was conducted across two iterations of the Genetic Algorithm. For the initial one, a starting population of 10 chromosomes instantiated with random values was created. A total of 800 generations were iterated over, with five children produced per generation. The test level had a time limit of 36 in-game seconds (~26 seconds in realtime), and a length of 300 blocks (~4800 pixels). The level's "seed" used by the level generation engine was 4 and the difficulty was set to 15. The program execution lasted over 20 hours.

After this initial iteration completed, five of the top-scoring agents were used as the starting population for the second iteration of the Genetic Algorithm. The level length was increased three-fold to a length of 900 blocks (~14400 pixels), the time limit set to 100 in-game seconds, the "seed" to 65, and the difficulty retained at 15. 320 generations were evaluated, again with five children produced for each generation. As this test level's length and time were much larger than the first iteration of the GA, the execution time prolonged to about 30 hours.

## Results

The technique of using a GA to evolve the SEF of a D* search allowed a system to generate an effective SEF in the absence of a priori knowledge about what makes one agent state more desirable than another. The results of this experiment demonstrate little to no direct correlations between individual weights and bot scores, implying a trial-and-error search for a human would be difficult and time-consuming.

For the initial iteration of the GA, over three thousand unique bots were evaluated over the course of 800 generations. 285 of those tied for the top score of 3955. An interesting note is that the first bot to score this amount was produced during the $11^{th}$ generation of the GA.

The weights used in the bot's SEF that the GA iterated on varied greatly. Figures 2 and 3 show typical scatter plots for the values of weights over the course of the GA's execution.

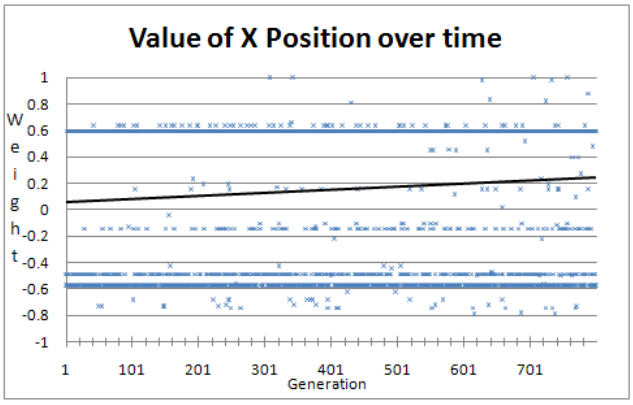


Figure 2

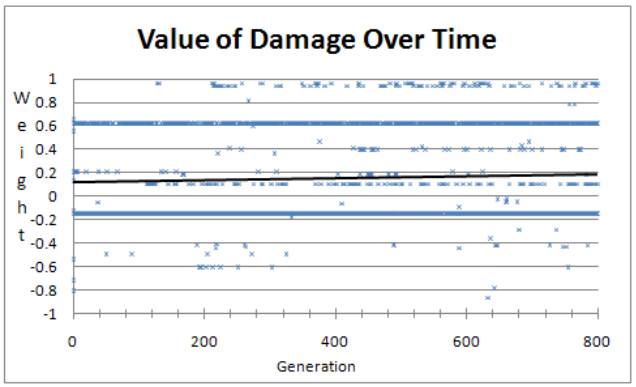


Figure 3

The weights on state-variables may appear to quickly converge to a few values and remain there. However, over time the amount of variance for any weight does not decline linearly. Figure 4 shows a plot of the amount of variation for each weight grouped by 50 generations. No r declination of the standard deviation among populations of weight values is present.

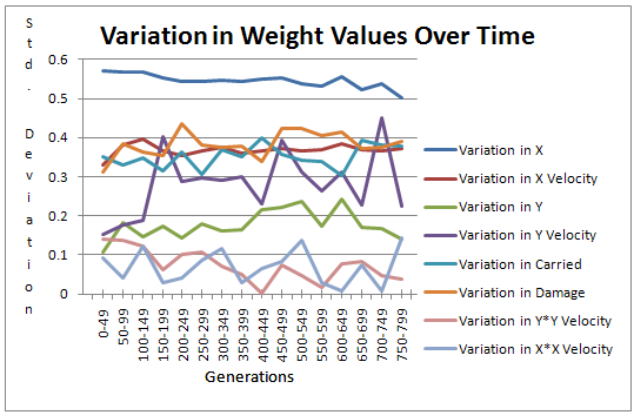


Figure 4

The scores that bots received likewise reached a local maximum early (generation 11), and were unable to improve thereafter (Figure 5).

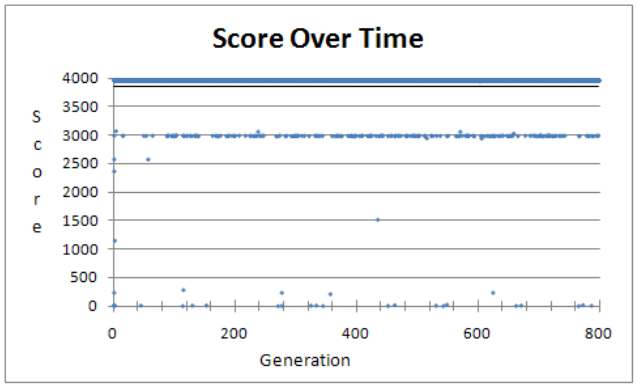


Figure 5

Upon examining the possible correlation between weight values and bot scores, a similar quandary is encountered where bots received top scores despite the weight values (Figures 6 and 7), save for the case of the weight on X Position multiplied by X Velocity (Figure 8). In the case of the value of the weight on the agent's X Position multiplied by the agent's X Velocity, a negative weight positively correlates to a higher score, and every single positive weight has a score of 0.0 or less (a negative score indicates the agent travelled backwards).

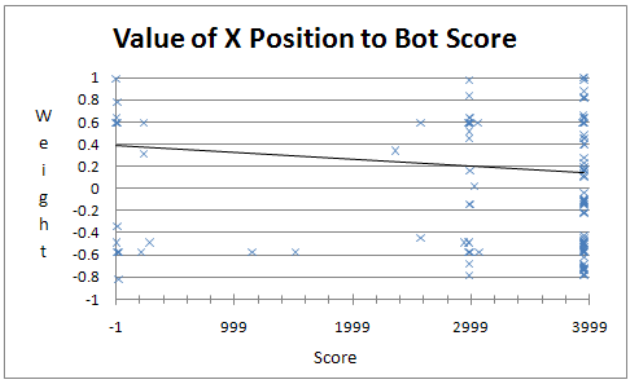


Figure 6

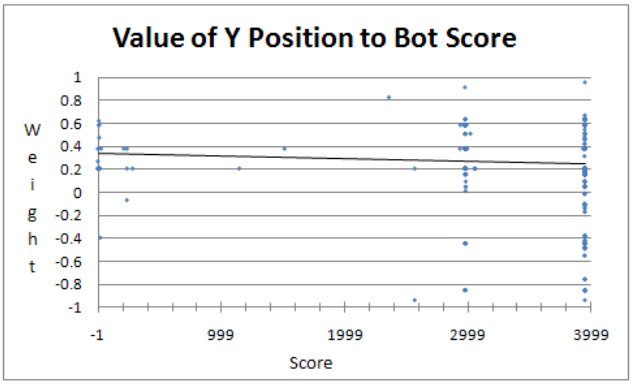


Figure 7

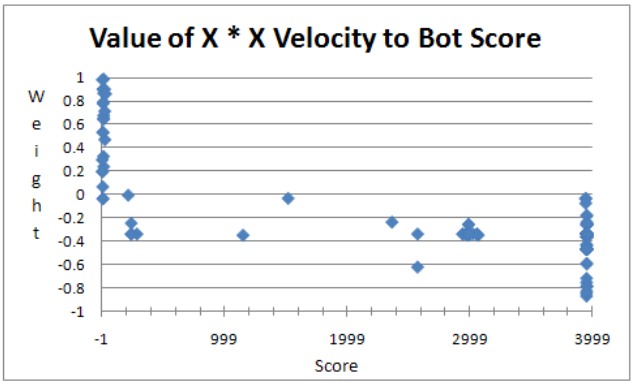


Figure 8

For the second iteration of the GA, similar results to the first iteration were obtained. However, only 3 bots out of a population of over a thousand shared the top score. Figure 9 presents the distribution of scores that bots received during the course of execution. Since the initial population of this iteration comprised top-scoring bots of the first iteration, it is understandable that so many bots scored so well so early, however a clear ceiling to the scores is visible, indicating the algorithm likely could not escape a local maxima.

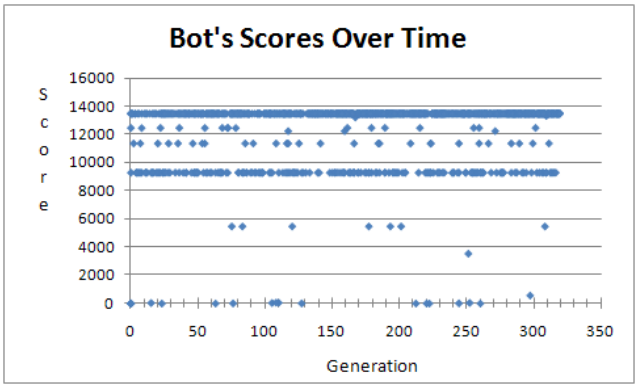


Figure 9

Figures 10 and 11 almost perfectly mirror Figures 6 and 7 in their distribution of scores for weight values on X Position and Y Position. Figure 12 likewise mirrors the data in Figure 8 that indicates negative weights on the agent's X Position multiplied by the Agent's X Velocity correlate to higher scores.

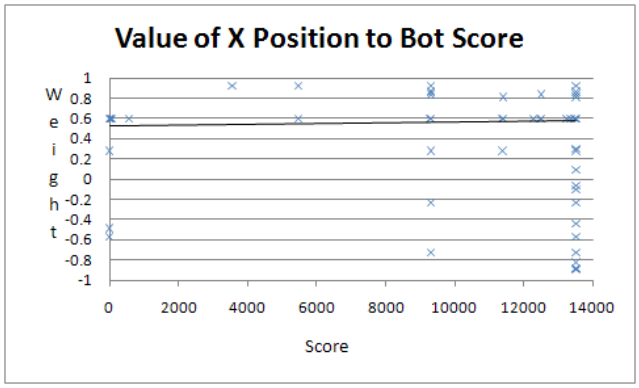


Figure 10

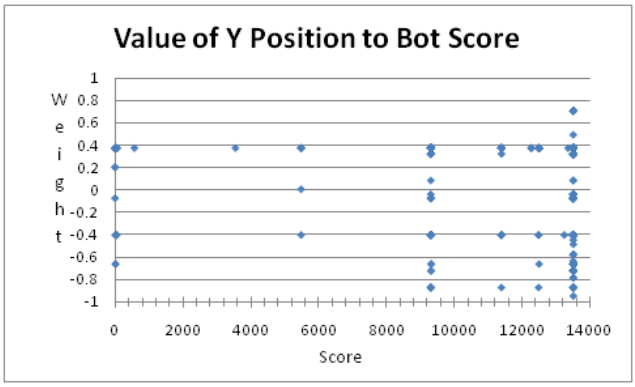


Figure 11

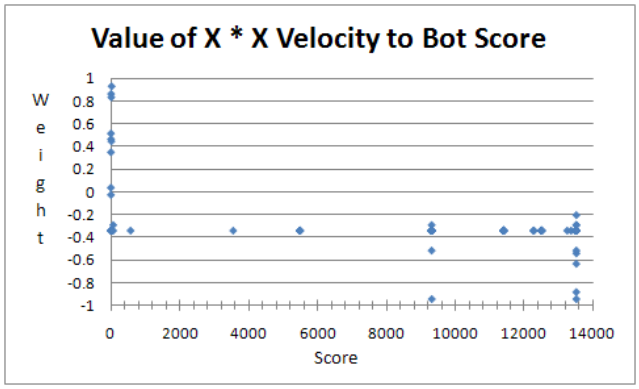


Figure 12

Although the weight values produced by the Genetic Algorithm for the top bots were distributed across the gamut of possible values, the end result was in fact bots

that performed roughly as well as Robin Baumgarten's bot that won the ICE-GIC competition two years in a row.

Table 1 displays a comparison of bot scores between Robin's Bot (AStarAgent) and our bot (AStarAgentEvolved) for a variety of levels whose difficulties were set to 15.

| Table 1. Comparison of scores for bots across a variety of levels | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| Agent | | | | | | | | | |
| | Level Seed | 4 | 40 | 65 | 100 | 112 | 216 | 325 | Total |
| AStarAgent | | 4450 | 4422 | 4420 | 4420 | 4470 | 4548 | 4470 | 31200 |
| AStarAgentEvolved | | 4452 | 4421 | 4420 | 4419 | 4468 | 4548 | 4469 | 31197 |

Table 1

## Conclusions

In this work, we presented the novel technique of using a genetic algorithm as an offline meta-search for an optimal static evaluation function to be used by the D* search of a real-time autonomous agent. The end results were Static Evaluation Function parameters that, upon use in the SEF for a real-time agent, enabled the agent to perform as well as the current best in its environment.

The fact that our agent performed as well as the current best is significant because we made very few assumptions about the valuation of agent states in a static evaluation function. That is, the algorithms presented in this paper automated this task. The implication is that similar techniques could be employed for autonomous agents in other, possibly real-world, environments with high confidence in the end result.

## Future Work

The work presented here has much potential for expansion. Future work should include utilizing parallel computing clusters like Beowulf to take advantage of the natural independence between the analyses of members in a population by the GA's fitness function, as well as the evaluation of nodes in the open list of the D* algorithm by the algorithm's SEF. This sort of capability will allow for not only a deeper D* search, but shorter generations in the Genetic Algorithm and therefore the ability to run the algorithm for more generations in the same amount of time.

Potential future work could also include employing pattern matching techniques to identify a discrete set of distinct scenarios an agent would encounter. An agent could then utilize a separate SEF for each scenario.

Under the notion of pattern-matching, even further future research would focus on generating probability tables for the likelihood of scenarios occurring after each other. Knowing the probability of a scenario to occur next would allow an agent to make an accurate prediction of an optimal path before receiving its next percept.

## References


Baumgarten, Robin. Infinite Super Mario AI. 9 September 2009. 8 February 2011 <http://www.doc.ic.ac.uk/~rb1006/projects:marioai>.

Hart, Peter E., Nils J. Nilsson and Bertram Raphael. "A Formal Basis for the Heuristic Determination of Minimum Cost Paths." IEEE Transaction of Systems Science and Cybernetics SSC-4, No. 2 (1968): 100-107.

Hill T, Lundgren A, Fredriksson R, Schiöth HB (2005). "Genetic algorithm for large-scale maximum parsimony phylogenetic analysis of proteins". *Biochimica et Biophysica Acta* **1725** (1): 19–29.

Koenig S. and Likhachev M. Fast Replanning for Navigation in Unknown Terrain. Transactions on Robotics, 21, (3), 354–363, 2005

Lucas, S., and Kendell, G. (2006). Evolutionary computation and games. *IEEE Comput Intell Mag*., pp. 10–18

Mitchell, M. (1998). An introduction to genetic algorithms. MIT Press.

Ramalingam G., Reps T., An incremental algorithm for a generalization of the shortest-path problem, Journal of Algorithms 21 (1996) 267–305.

Russel, Stuart J. and Peter Norvig. Artificial Intelligence: A Modern Approach. Upper Saddle River: Prentice Hall/Pearson Education, 2003.

Zhang, J., Lo, W.L., and Chung, H. (2006).Pseudocoevolutionary Genetic Algorithms for Power Electronic Circuits Optimization. *IEEE Trans Systems, Man, and Cybernetics*, **36C** (4), pp. 590–598.