Proceedings of the 2007 IEEE Symposium on
Computational Intelligence and Games (CIG 2007)

Using a Genetic Algorithm to Explore A*-like Pathfinding Algorithms

Ryan Leigh, Sushil J. Louis, and Chris Miles
Evolutionary Computing Systems Lab
Dept. of Computer Science and Engineering
University of Nevada, Reno
{leigh, sushil, miles}@cse.unr.edu

***Abstract*— We use a genetic algorithm to explore the space of pathfinding algorithms in Lagoon, a 3D naval real-time strategy game and training simulation. To aid in training, Lagoon tries to provide a rich environment with many agents (boats) that maneuver realistically. A*, the traditional pathfinding algorithm in games is computationally expensive when run for many agents and A* paths quickly lose validity as agents move. Although there is a large literature targeted at making A* implementations faster, we want believability and optimal paths may not be believable. In this paper we use a genetic algorithm to search the space of network search algorithms like A* to find new pathfinding algorithms that are near-optimal, fast, and believable. Our results indicate that the genetic algorithm can explore this space well and that novel pathfinding algorithms (found by our genetic algorithm) quickly find near-optimal, more-believable paths in Lagoon.**

Keywords: Genetic Algorithms, A*, Pathfinding

I. Introduction

Navigating safely in a game or training simulation is a challenging problem. Complexities such as moving obstacles, physics, and a changing environment can cause danger for entities attempting to find safe passage. As an agent AI testbed, we are developing Lagoon, a real-time 3D naval combat and training simulation. For training purposes, a naval instructor would use Lagoon to present lessons to students on how to identify and manage risks on the sea or in the harbor. The agents in the simulation need to act like humans similar scenarios for lessons learned in the training simulation to translate to real-life. Believable pathfinding is a critical component for creating these believable agents [1], [2].

Our previous work used a Genetic Algorithm (GA) to tune behavior-based controllers [3], [4] to maneuver through a 2D naval simulation [5]. The agent in the 2D simulation provided a baseline for future work in Lagoon. While GA-tuned agents performed well, the reactive obstacle avoidance of agents in the 2D simulation had difficulties passing through narrow channels. In this paper, we investigate pathfinding algorithms to provide high-level planning for the agents so that they can be better employed in Lagoon.

Pathfinding algorithms are a subset of net search algorithms [6]. Nets consist of nodes, links connecting nodes, and application-specific link labels. These three parts combine to give application specific semantics to networks. Nets can be used to describe many things, from the procedures robots use to operate, to relatedness of ingredients in a recipe. Searching these example nets can produce the steps necessary for the robot to overcome an obstacle or find new recipes. Pathfinding is often used in the context of games where nets represent the game map. Our research can apply to any net search problem, but in this paper, we focus on pathfinding in games.

A* is the *de facto* pathfinding algorithm for games that combines dynamic programming and an admissible distance heuristic to trim search. It typically takes longer to complete than one time cycle of the game. By the time a path is found, the position of entities could be different or the environment may have changed, rendering the path inaccurate or even hazardous. Therefore, we would need to run A* as often as possible for every entity, a heavy computational burden. The paths may also look unrealistic. A* may choose "shortcuts" that a human would never consider, play it safe when a human would take a risk, or find paths that can feel rough or too angular as shown in Figure 1.

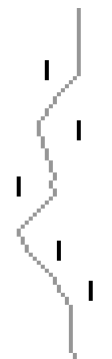

Fig. 1. Results from an A* search

We seek to resolve these time and believability issues by using a GA to explore the space of pathfinding algorithms. We intend to address the following questions: How can A* be modified to find near-optimal paths quicker? What heuristics can lead to these paths? How can A* find believable paths? Can search progress if we limit processing time for the algorithm to short periods? Our results show that our GA-Manufactured Maneuvering Algorithm (GAMMA) can make significant performance gains over A* in time taken to find

1-4244-0709-5/07/$20.00 ©2007 IEEE

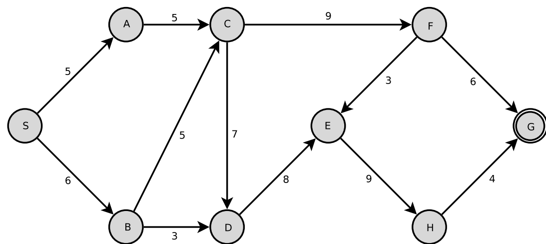

Fig. 2. A net of nodes and edges

a path while staying near-optimal and believable.

This paper is organized as follows: the next section provides background on the type of search we use as well as related work. Section III describes our 2D simulation. Sections IV and V describe the parameters of the GA and the experiments we performed. Section VI contains the results. We have discussion and future work in Section VII and finally summarize in Section VIII.

II. Background

A. The A* algorithm

A* is the *de facto* pathfinding algorithm that, under the right conditions, guarantees to find a path with the lowest possible traversal cost. A* and other pathfinding algorithms work on a net. Figure 2 shows an example of a net, a collection of nodes connected by links. Each link has a value (shown as an integer) which is the cost incurred by traversing that edge. Pathfinding algorithms try to find a sequence of nodes from a start node to a goal node having the lowest traversal cost.

A* works by building all paths that lead to the goal, one path at a time and starts by ranking each partial path currently in consideration via the following formula [6], [7]:

$$f(x) = g(x) + h(x) \tag{1}$$

where x is some partial path, f is the estimated final cost of path through x to the goal, g is the known cost to traverse to x, and h is the estimate of remaining distance to the goal from the end of x. The guarantee of lowest cost paths only applies if the remaining estimate to the goal is an underestimate. A* then uses the procedure in Figure 3 to find the optimal path.

- Form a one-element queue consisting of a zero-length path that contains only the start node.
- Until the first path in the queue terminates at the goal node or the queue is empty,
  - Remove the first path from the queue; create new paths by extending the first path to all the neighbors of the terminal node.
  - Reject all new paths with loops.
  - If two or more paths reach a common node, delete all those paths except the one that reaches the common node with the minimum cost.
  - Sort the entire queue using Formula 1, with the least-cost paths in front.
- If the goal node is found, announce success; otherwise announce failure.

Fig. 3. Pseudo-code for A* algorithm [6]

B. Heuristics

The performance of A* relies heavily upon the estimate of remaining distance to the goal, called a *heuristic*. For A* to find the shortest path, the heuristic must underestimate the remaining distance. When A* finds a possible path to the goal, if all other partial paths' lengths plus the remaining estimated distance measures more than another path found to the goal, then that path found is optimal. The optimal path can never have a shorter length than an underestimate. If the heuristic can make overestimates, then A* may discard a shorter path because the partial path's current distance plus the overestimate would make it longer than the path found to the goal.

We do not need an optimal path, just a safe, near-optimal path, so an overestimate may provide quicker results. It could push the search into areas that may have a higher cost to travel through, yet take less time to explore or provide more believable paths. With this in mind, we used four distance heuristics, three of which are illustrated in Figure 4.

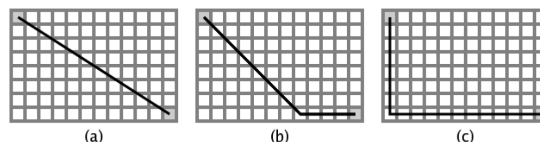

Fig. 4. Heuristics: (a) Euclidean, (b) Diagonal Shortcut, (c) Manhattan Block

As the figure shows, we divide the world into unit cells. We measure the distance of moving straight up, down, left, or right as 1 unit and a diagonal move as 1.4 units. The diagonal of a unit square measures $\sqrt{2}$ which approximately equals 1.4. Many computers operate faster with integers than floating point numbers, so we multiply the unit distances by 10 to get integer constants — 10 and 14. All distances become 10 times longer than their actual distance but still compare similarly when measured relative to each other.

Euclidean distance, our first heuristic, is the straight line distance between two points but is computationally expensive as each calculation involves two powers and a single square root. Equation 2 shows the formula to measure the distance where (x_{path}, y_{path}) is the point at the end of the path and (x_{goal}, y_{goal}) is the goal. This heuristic guarantees to underestimate, because the shortest distance between two points is a straight line. We can make it overestimate by omitting the square root while also reducing the run-time. This is our second heuristic.

$$\text{Distance} = \sqrt{(x_{path} - x_{goal})^2 + (y_{path} - y_{goal})^2} * 10 \quad (2)$$

The diagonal shortcut heuristic measures at a 45 degree angle until it reaches the same axis as the goal and then measures the remaining distance in a straight line. The diagonal shortcut method takes less computational time than Euclidean distance because it only uses addition and subtraction. Equation 3 shows the formula if the vertical distance (dy) between the path end and goal is greater than the horizontal distance (dx).

$$\text{Distance} = 14 * dy + 10 * (dx - dy) \quad (3)$$

We use the Manhattan block heuristic as our last heuristic. It sums the vertical and horizontal distance traveled to get to the goal. Manhattan Block get its name because one might move that way on city blocks. Of the four heuristics, this is the quickest, but it can overestimate if the simulation allows diagonal moves as ours does.

C. A* modifications

By the time A* finds a complete path to the goal, the environment may have changed. To help prevent this, we only allow our GAMMAs to plan a limited distance by limiting search to short time phases. For this research, each GAMMA can only expand a limited number of paths for each phase and at the end, the start location is moved to the end of the best path found in that phase. Pathfinding then begins anew. Phases are based on the number of nodes expanded and not on an actual time limit so that we can repeat the experiment. Running the same GAMMA over different runs may result in different final running times because of background processes running along with the GA on the same machine. The GAMMA's run-time factors heavily into its fitness, thus the same GAMMA may receive a different fitnesses over different runs of the GA. By using the number of paths expanded to measure run-time, we have a machine independent measurement that is still a fair representation of actual run-time.

Phasing can cause infinite loops because it can put the algorithm into an area that it cannot find a path out of within the time limit. We introduce a global time limit over all phases and individuals that take longer than the allotted time get penalized.

We also introduce the idea of end-path sampling. Rather than sum the cost of the entire path, the GAMMA could only use the last few nodes. The GA determines whether or not to use sampling, how many nodes to use, and a possible multiplier. This also means the GAMMA could sample no nodes at all and judge paths based solely on the heuristic.

Lastly, we give the GA two more ways to change A*: how new paths are added and in what order. First, newly expanded paths can either be added to the top or bottom of the list of previously expanded paths. Second, the order that paths get expanded can be reversed, so that new paths get added to the list first to last or last to first. Both modifications can change which path gets chosen when choosing between paths with the same evaluation cost.

D. Risk zones

Risk zones add additional cost to nodes around the entity. This additional cost pushes the GAMMAs away from these risk zones. Each entity has 16 arc zones round the perimeter of the entity, with the first arc starting at the front as shown in Figure 5. Each zone has a radius and a risk value which is the additional cost a path incurs by moving through a node in that zone. A shorter path that moves through a risk zone will likely have a higher cost than a longer path that moved around the risk zone. The GA tunes these zones to shape the GAMMAs' paths closer to the human-like path. For example, longer zones in the front while shorter zones in the back would allow the GAMMA to pass closer to the rear of the entity than the front without incurring additional risk. If a path lies in a risk zone, the additional cost due to risk equals $maxDistance - currentDistance$, where $currentDistance$ is the distance from the path node to the entity and $maxDistance$ is the furthest a zone can extend. To help round out the risk zones, for example if adjacent zones very greatly in length, we average zone lengths using the following formula:

$$zone_{i,final} = \frac{zone_{i-i} + zone_i + zone_{i+1}}{3} \quad (4)$$

where i is a particular zone. Only initial values for surrounding zones are used for the average. Figure 6 shows the zones as they appear in the simulation.

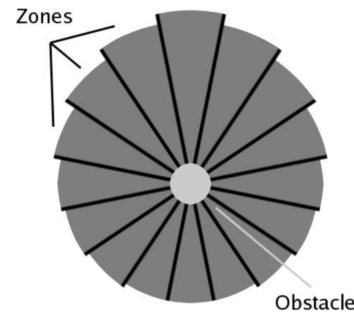

Fig. 5. Diagram of zones around an obstacle

E. Related Work

Many methods have been used in the past to speed up A*. Iterative Deepening A* (IDA*) uses no storage and uses a depth-first, iterative-deepening search of the net [8]. IDA* works by doing depth-first search, calculating the path cost using Formula 1. The search continues recursively until a threshold for f is met or the goal is found. If the threshold for all paths is met and no goal is found, IDA* increases the threshold and search starts anew. Bjornsson, et. al., expanded on IDA* with Fringe Search [9]. Fringe Search keeps memory of leaf nodes as it explores and on the next iteration starts exploration from those leaf nodes.

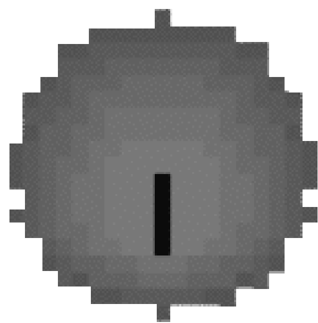

Fig. 6. Zones from Figure 5 as displayed in the simulation

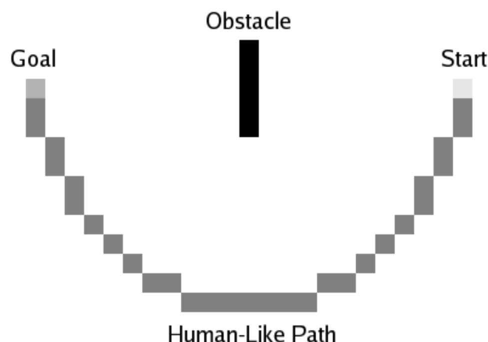

Fig. 7. A sample map of start location, end location, and human-like path

Our method shares a similarity with Fringe Search with our phasing. We limit the search depth by only allowing expanding a certain number of paths. Where Fringe Search keeps all leaf nodes and will find the optimal path, we only seek near optimal paths and thus greedily choose only the best path. We also introduce a hard cap for paths expanded meaning our search space can be kept small. Our search space can also be easily expanded by adding the features of IDA* and Fringe Search. The GA can then find new combinations of features not yet explored.

GAMMA uses more state than IDA* and Fringe Search by keeping the complete sorted list of all open paths like A*. GAMMA can overcome this overhead by using end-path sampling. Only paths that end in low risk zones closer to the goal than other paths get expanded. This helps to keep the open list small. The related search algorithms also use the same heuristics as A*, so they too will lack the human-like considerations we have by adjusting risk zones and will lose believability.

III. SIMULATION

We created a simple simulation to provide quick evaluations for GA exploration of the search space because we first seek a proof-of-concept. The simulation creates the environment from two parts: an image map and an entity description file. The image describes the search space where each pixel is a node in the search net. The color of each pixel provides additional context for the node: white is an open node, black is a closed node, green is the start location, red is the goal, and a blue pixel shows the human-made path (Figure 7). The entity description file lists each entity's position and heading in the environment and the simulation uses this information to construct the risk zones.

To evaluate an individual, the simulation runs each GAMMA (individual) over a number of different maps, each with a different human-made path. The GA evaluates GAMMAs based on two objectives: the difference from the human-made path and the cost of the found path, both of which are minimized. The simulation calculates the difference from the human-made path by summing the distance between each node of the found path and the closest node on the human-made path. If the found path follows the same route as the human-made path, then the difference would be zero. The simulation calculates the cost of the found path by adding the traversal cost of the path plus the risk value of every node on the path.

TABLE I
CHROMOSOME PARAMETERS

Bits	Value	Description
2		Heuristic used
-	0	Euclidean distance
-	1	Euclidean distance squared
-	2	Manhattan block distance
-	3	Diagonal shortcut distance
1	Bool	Use end-path traversal cost
2	0-3	End-path sample depth
2	1-4	End-path traversal cost multiplier
1	Bool	Add new path to top (True) or bottom (False) of open path list
1	Bool	Add new paths in reverse order if True
4	1-16	Zone 1 range
4	1-16	Zone 2 range
...		
4	1-16	Zone 16 range

Lastly, the simulation handles the changing of phases. It clears all bookkeeping the pathfinding algorithms have made up to that point and moves the starting location. Each GAMMA expands 100 paths per phase.

IV. METHODOLOGY

The genetic algorithm uses single point crossover, bitwise mutation, and roulette wheel selection. The GA doubles the population with offspring and keeps the best n, where n is the population size, for further processing [10]. We set the crossover rate at 1.0. Given that the GA evaluates the parents with the offspring, we do not worry about losing valuable information from the parents if the offspring performs worse. We set the mutation rate at .03 or approximately two mutations per chromosome. Each GA is run for 40 generations with a population size of 200. Table I gives the parameters for each 73 bit chromosome. Results are averaged over 20 runs.

V. Experiment

We ran each GAMMA over three separate maps as shown in Figure 8. All entities in the three maps face forward. The path on the first map establishes that moving around the rear of the ship is safer than the front. The last two maps use the same configuration of entities but with switched starting and stopping locations and different human-made paths. The different paths serve three purposes: to establish that moving between entities is acceptable, to prevent the algorithm from just navigating around all the entities, and to prevent the GAMMA from over-specializing.

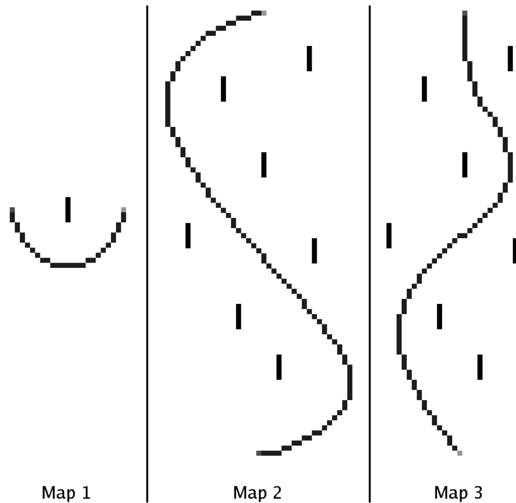

Fig. 8. Maps used in the experiment

The GA evaluates an individual by evaluating its performance based on each objective separately. The GA sums an individual's performance for a given objective over each map and normalizes the summations so that the GA can later compare objectives on equal footing. We seek an algorithm that can minimize both objectives, but because GAs work by maximizing values, it must invert each objective value so that low values become high values. Next, the GA scales the normalized, inverted values such that the highest value is 1.2 times the average of the objective value over all individuals. We do not want individuals with significantly high objective values to dominate selection. Lastly, the GA sums each final objective score to produce the final fitness for an individual. Equation 5 gives the final form of this calculation where a and b are values used for linear fitness scaling. a and b are chosen to ensure the highest fitness is 1.2 times the average and i is the objective [11].

$$f_{final} = \sum (a_i \frac{f_{max_i} - f_i}{f_{max_i} - f_{min_i}} - b_i) \quad (5)$$

VI. Results

We ran the GA using 20 different seeds and in Figure 10 we show an example result. The "clouds" around each entity are the risk zones. The GAMMA found the light gray path in the figure and while it is similar to the path specified by the user, it is still not exactly the same. It moves through the right areas, but acts "jittery." The GAMMA shown in the figure sampled only the last point in the path with a multiple of three and used the Euclidean heuristic. The pseudo-code for this GAMMA is given in Figure 9. Key parts that are different from the A* pseudo-code are shown in the inner boxes. The selection of the Euclidean heuristic seemed to be a common feature among all GAMMAs. While other heuristics take less CPU time to run, this is not accounted for by our time measurement of using paths expanded. While other heuristics may gain favor by using a CPU run-time measurement, the GA instead chooses the heuristic that gives the best underestimate. Whether this is coincidental or a direct result of the paths expanded run-time measurement is left to future work.

- Form a one-element queue consisting of a zero-length path that contains only the root node.
- Until the first path in the queue terminates at the goal node or the queue is empty,
  - If number of paths expanded is equal to the number of paths to expand per phase,
    - Choose best path in queue.
    - Clear queue and add best path to queue.
  - Remove the first path from the queue; create new paths by extending the first path to all the neighbors of the terminal node.
  - Reject all new paths with loops.
  - If two or more paths reach a common node, delete all those paths except the one that reaches the common node with the minimum cost.
  - Sort the entire queue by the sum of cost of the last node in the path multiplied by three and the Euclidean distance from that node to the goal, with the least-cost paths in front.
- If the goal node is found, announce success; otherwise announce failure.

Fig. 9. Pseudo-code for selected GAMMA algorithm. Key changes to the A* algorithm are boxed

To provide a baseline, we ran A* with hand-tuned risk zones. A* ran from start-to-finish in one run and with time phases like the GAMMAs. A* used the Euclidean heuristic. Figure 11 shows that neither start to finish nor phased A* fits the human-like path very well while the GAMMA results are much closer.

Although the GA can create good GAMMAs, there can still be a wide variation in results. Figure 12 show results from a different GAMMA. The GAMMA does not fit the human-like path very well in the first map. Label (a) shows where the GAMMA takes a 90 degree right turn. While physics and path smoothing would soften this turn, it is not

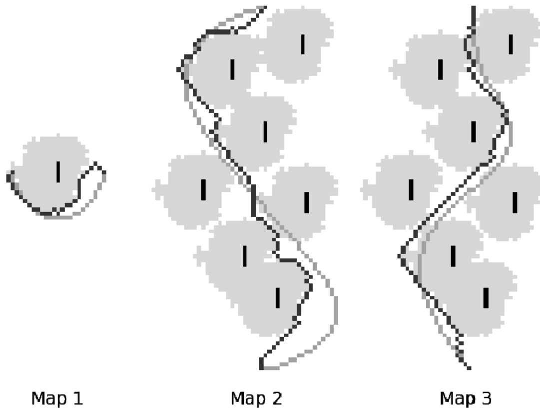

Fig. 10. Sample results for GAMMA found paths

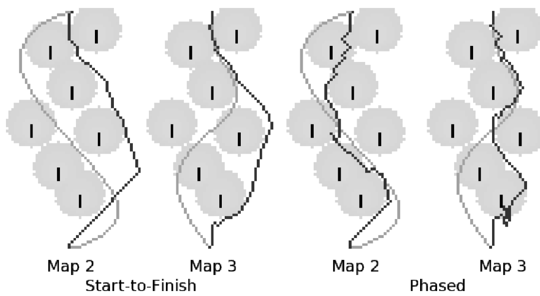

Fig. 11. Paths found by A* when: (a) start to finish, (b) in phases

believable that a human would make this decision. Label (b) shows where the boats must zig-zag. The same zig-zagging appears at the end of the second map. Again, physics will dampen this behavior, but there could still be some unusual fishtailing.

Table II shows the numerical results from Figure 10. The GAMMA produces a final path 1271% faster than traditional A* and 353% faster than phased A*. Figure 13 shows the results of the GA for each objective averaged over the 20 runs.

To see how well the selected GAMMA worked, we created two maps and ran the selected GAMMA on them. These maps have no human-like path as they represent the new situations the GAMMA could face in Lagoon. Figure 14 shows the paths found by the different algorithms and Table III shows the number of cycles needed to find these paths. Although the GAMMA does find safe paths, they lack the smoothness we were seeking and additional work needs to be done to generalize the GAMMAs. These additional results show that there is some over-specializion to the maps on which the GAMMAs are evaluated. This is not necessarily bad as the GA can optimize the GAMMA for a particular map set so that users would not have to do that themselves.

As a bridge to future work, we took the selected GAMMA and implemented it in Lagoon. Using the same configuration of entities, the pathfinder finds the same path as the 2D simulation and the small boat is able to follow the path as

TABLE II
FINAL RESULTS

	Cycles (% Improve)	Path cost (% Improve)	Closeness (% Improve)
GAMMA	432	2865	506
A*	5924 (+1271%)	2600 (-9%)	2087 (+312%)
Phased A*	1958 (+353%)	2878 (+0.5%)	1659 (+234%)

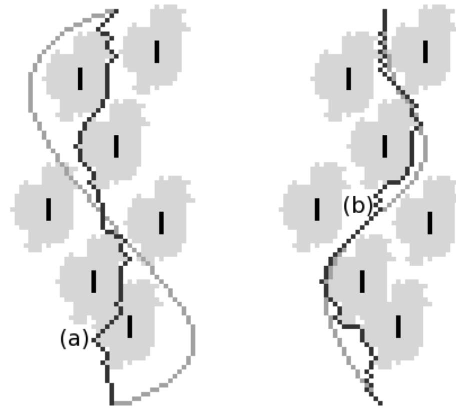

Fig. 12. More results from a different GAMMA with problems areas marked (a) and (b)

shown in Figure 15. Lagoon features more realistic physics and motion so that we can find GAMMAs more applicable to reallife scenarios.

VII. Discussion and Future Work

The resulting GAMMAs find paths quicker than A* and fit closer to the human-made paths while not costing much more than A*-found paths. Differences arise from both risk zone tuning and end path sampling. The algorithm shown in Figure 10 sampled only the last node of a path and multiplied that value by three. To improve generality and robustness, additional maps could be added.

Although results may be promising, there still exist areas for improvement. First, the environment needs to become dynamic. Dynamic environments can invalidate paths, so limiting the time that algorithms have to search forces the algorithms to constantly find new paths without wasting time working on distant paths.

Second, traversal costs do not model actual motion. It costs just as much to move forward and diagonal as it does to move backwards and diagonal, whereas, an actual ship would need to spend more time turning to face the back-facing diagonal than the front-facing diagonal. If a path suggests a maneuver that the entity cannot accomplish, then it may stray from

TABLE III
NEW MAP RESULTS

	Cycles
GAMMA	413
A*	2812
Phased A*	3718

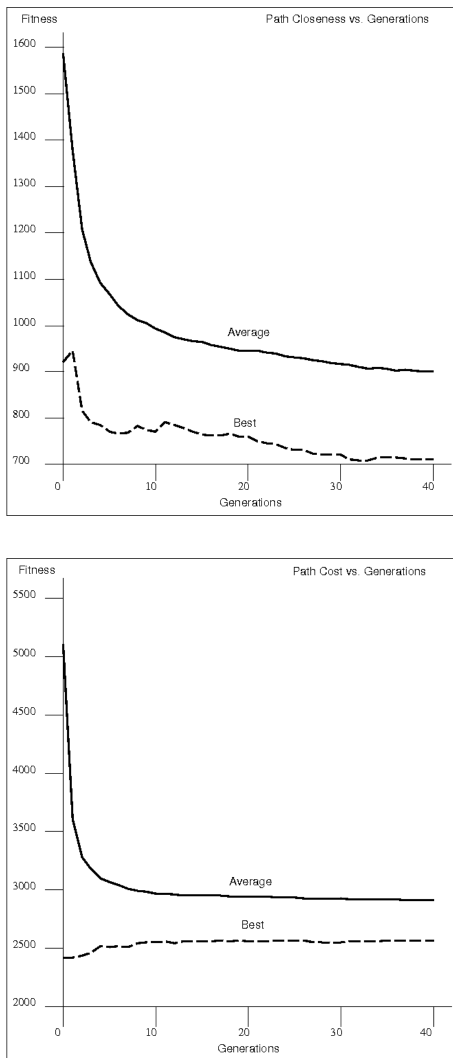

Fig. 13. Closeness objective value over time and cost objective value over time for GAMMA

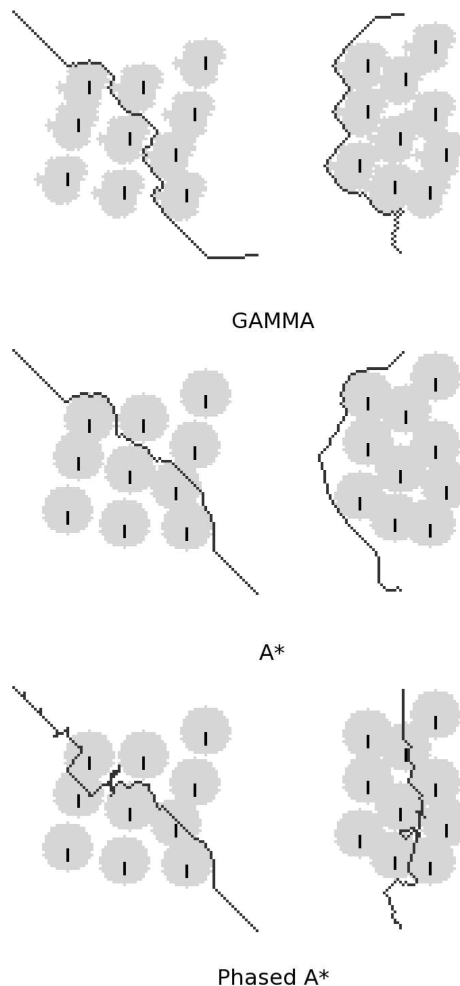

Fig. 14. Paths found on two separate maps. Each column is a map and results from each pathfinder are in the rows: GAMMA, A*, Phased A*

the path or worse — crash. The pathfinder should apply an approximation of physics when choosing the next node to expand, because the net is an approximation of the environment. Nets do not model continuous movement or time; they provide a snapshot of the environment. An additional GA may be needed to tune the physics approximation model so that the results from the 2D simulation work properly with Lagoon's physics model.

Third, it is not surprising that the resulting GAMMAs between runs of the GA lack consistency. A GAMMA from one run may perform much better than a GAMMA from different run. Currently, a human must explore the results of the different GA runs to hand select the best GAMMA. Automating this process of choosing the best GAMMA would ensure that less intervention would be needed by the user. Either the GA or fitness function could be modified so that the GA produces more consistent results or the GA could run many times and the highest fitness scoring GAMMA among all runs would be chosen.

In section VI, we showed that the GAMMAs seem to specialize to the maps on which they were trained. To provide more generalized and robust GAMMAs, we want to introduce co-evolution into our research. The model of co-evolution used by Rosin and Belew takes two or more populations and evolves them simultaneously using coupled fitness [12]. For example, using two populations, one population tests the other population. The tested population receives a fitness and evolves, then the two populations switch roles and the tester becomes the tested. The two populations constantly challenge each other, overcoming obstacles the other side presents while coming up with new obstacles of its own. Co-evolution may work well with pathfinding. The GAMMAs compete against a map planner which evolves to generate increasingly difficult maps that the GAMMAs must navigate. The GA also has a harder time over specializing

Fig. 15. The GA-found pathfinding algorithm in Lagoon

the GAMMAs because the GAMMAs must constantly work on different maps. The is a problem though because the automated map planner cannot specify human-like paths on its own and the human-like paths are a critical component in making GAMMAs that produce believable paths.

VIII. Summary

We used a genetic algorithm to explore the space of pathfinding algorithms. We sought to find path finding algorithms that could outperform A* by taking less time to find a path that more closely matches paths made by a human. A GA modified components of a baseline A* algorithm and tuned risk zones around entities to find alternative algorithms. The simulation also limited how long our GA-Manufactured Maneuvering Algorithms (GAMMAs) could run before they were reset with a new starting location. One of the GAMMAs performed 1271% faster than a traditional A* but testing under a dynamic environment with a better traversal model cost will be needed before we can use these algorithms.

Acknowledgments

This material is based upon work supported by the Office of Naval Research under contract number N00014-05-0709.

References

[1] B. D. Bryant, “Evolving visibly intelligent behavior for embedded game agents,” Ph.D. dissertation, Department of Computer Sciences, The University of Texas at Austin, 2006.

[2] R. Miikkulainen, B. D. Bryant, R. Cornelius, I. V. Karpov, K. O. Stanley, and C. H. Yong, “Computational intelligence in games,” in *Computational Intelligence: Principles and Practice*, G. Y. Yen and D. B. Fogel, Eds. IEEE Computational Intelligence Society, 2006.

[3] R. C. Arkin, *Behavior-Based Robotics*. CA: MIT Press, 1998. [Online]. Available: http://www.usc.edu

[4] M. J. Matariĉ, “Behavior-based control: Examples from navigation, learning, and group behavior,” *Journal of Experimental and Theoretical Artificial Intelligence*, vol. 9, no. 2-3, pp. 323–336, 1997, journal of Experimental and Theoretical Artificial Intelligence.

[5] R. E. Leigh, T. Morelli, S. J. Louis, M. Nicolescu, and C. Miles, “Finding attack strategies for predator swarms using genetic algorithms,” in *Proceedings of the 2005 Congress of Evolutionary Computation*, September 2005.

[6] P. H. Winston, *Artificial Intelligence*, 3rd ed. Addison-Wesley, 1992, ch. 5.

[7] P. E. Hart, N. J. Nillson, and B. Raphael, “A formal basis for the heuristic determination of minimum cost paths,” *IEEE Transactions on Systems Science and Cybernetics*, vol. 4, no. 2, pp. 100–107, 1968.

[8] R. E. Korf, “Depth-first iterative-deepening: an optimal admissible tree search,” *Artif. Intell.*, vol. 27, no. 1, pp. 97–109, 1985.

[9] Y. Bjornsson, M. Enzenberger, R. C. Holte, and J. Schaeffer, “Fringe search: Beating a* at pathfinding on game maps,” *IEEE Computational Intelligence in Games 05*, p. 125132, 2005.

[10] L. J. Eshelmann, *The CHC Adaptive Search Algorithm: How to Have Safe Search When Engaging in Nontraditional Genetic Recombination*. Morgan Kauffman, 1990.

[11] D. E. Goldberg, *Genetic Algorithms in Search, Optimization, and Machine Learing*. Addison Wesley Longman, Inc., 1989.

[12] C. D. Rosin and R. K. Belew, “New methods for competitive coevolution,” *Evolutionary Computation*, vol. 5, no. 1, pp. 1–29, 1997. [Online]. Available: citeseer.ist.psu.edu/rosin96new.html