Abstract—This paper describes a genetic algorithm for solving the traveling salesman problem (TSP) for autonomous navigation. The method is applied to autonomous underwater vehicles for efficient path planning during underwater mine inspections, sponsored by the Office of Naval Research. This method is significantly easier to implement and much more extensible to real world variants of TSP, e.g. problems incorporating currents, limited turning radius, limitations in depth changes, etc., than other, more efficient, approaches. A specific case study demonstrates a variation accounting for constant currents. Performance is compared against existing behaviors for path planning implemented in the Mission-Oriented Operating Suite (MOOS). The results show that the genetic algorithm performs significantly better than the approach currently implemented in MOOS and successfully accounts for factors such as currents.

I. INTRODUCTION

Autonomous underwater vehicles (AUVs) are increasingly desirable tools for performing underwater missions. AUVs often perform tasks in environments which are too dangerous for human divers, or which are otherwise inaccessible. Further, employing AUVs can be significantly less expensive than hiring divers or manned submersibles. Finally, underwater vehicles exhibiting high levels of autonomy are more useful and more cost effective for performing difficult, time consuming, or redundant tasks. Many missions require traveling between multiple objective locations to perform some task, in which case AUVs must optimize the order in which objective locations are visited. Poor optimizations will result in wasteful use of limited battery resources. The goal of this research is to solve this problem in a practical way.

The Office of Naval Research (ONR) is tasked with finding a method for locating, identifying, and neutralizing underwater mines. The application of AUV technology to mine countermeasure (MCM) missions should allow the ONR to perform this critical task with lower risks and costs than previously possible. The strategy they are considering involves several automated vehicles. Surface-based scanners generate maps of mine-like objects (MLOs). The locations of these MLOs are sent to the AUVs as they are located in real-time. Each AUV is then responsible for inspecting each MLO it is assigned in order to improve confidence of detection or to disarm the object. Generally, MLOs are discovered much more quickly than they can be inspected, so AUVs have several inspection locations to consider at any given time. A significant amount of battery resources may be expended traveling between these locations if the ordering of visitation is random or naively selected. This suggests a need for a control module which optimizes the ordering selected by the AUV for visiting MLOs to reduce the total distance traveled and ultimately the energy cost sustained during the course of a mission.

The research presented here describes an algorithm implemented on the AUV for planning the order of inspections. The resulting algorithm is useful in any scenario requiring inspections at multiple locations during a single mission. It could be employed in the inspection of underwater wreckage sites or for taking samples that measure toxicity levels, for example. Our results show that the algorithm is capable of providing better optimizations than a nearest-neighbor (NN) heuristic, which is the algorithm currently supported within the Mission-Oriented Operating Suite (MOOS). Furthermore, we show that, unlike more traditional TSP algorithms, the presented algorithm can optimize with respect to arbitrary costs and constraints that affect the optimal path of an AUV.

II. BACKGROUND

A. Application Platform

To support the advancement of AUV technology, modular software control systems have been developed, enabling standardized development of shared modules, rapid prototyping, and enhanced computer simulation. The ONR is supporting the use and development of MOOS as the framework of choice for AUV research.

The MOOS application framework is a series of applications and libraries which collectively provide control to an autonomous vehicle. By creating independent modules and libraries in C++, authors can share and collaborate on a consistent platform for development. The platform is centralized around an event-driven key-value store known as the MOOS Database (MOOSDB). A module can subscribe to receive updates on keys in the database such as a vehicle’s speed, available energy, or serial communications data. Multiple independent applications use this central database to communicate,
simplifying interaction for a module author considerably. A complete overview of MOOS is [1].

A control module known as the Helm-IvP performs a multi-objective optimization algorithm to balance multiple concerns of a mission with a series of predefined behaviors. A mission could include instructions to follow a series of waypoints (using the waypoint behavior module) and maintain a constant depth. The Helm-IvP module will optimize for these behaviors simultaneously and send control signals for directing the autonomous vehicle as it travels. Parameters of behaviors can be altered in real-time by running MOOS modules, such as updating a list of waypoints during the course of a mission or changing the speed of a vehicle to improve energy efficiency. An overview of available behaviors is provided in [2].

B. Traveling Salesman Problem

In its simplest form, finding an efficient ordering of points for inspection is equivalent to the well-known problem in computational mathematics known as the Traveling Salesman Problem (TSP). Finding the optimal solution to this problem is NP-complete, meaning that no known efficient solution exists. The total number of possible tours grows exponentially as the number of locations to be investigated increases. Additionally, practical considerations such as ocean currents, maximum turning radius, and other factors make the problem even more complicated.

C. Solution Approaches

Several methods exist for solving the TSP. The methods can be categorized as those that solve the problem exactly and those which find approximate solutions. Every exact method developed to date has exponential complexity in the worst case, and it is generally believed that no exact algorithm will ever be discovered that can run in less than exponential time with finite resources. For this reason, the following discussion focuses on approximate algorithms.

There are two methods generally employed in algorithms which find approximate solutions to the TSP. Constructive methods build solutions from scratch one component at a time. Iterative improvement methods begin with a random tour, and then incrementally improve upon it to achieve better solutions. Some algorithms use a combination of these techniques.

1) Nearest-Neighbor Algorithm: The classic constructive method for solving the TSP is the nearest neighbor (NN) heuristic algorithm. This algorithm picks a starting location at random and then finds the nearest location to that position. The pair becomes the first path along the tour. The algorithm continues from the new location by finding the nearest location to that position that has not already been made part of the tour. This continues until the last location is added to the tour. This algorithm requires \( n - 1 \) iterations, where \( n \) is the number of locations. In each iteration, between 1 and \( n - 1 \) locations are examined for proximity to the current location, so the algorithm requires \( O(n^2) \) time to run. The start of the tour is fairly efficient, but as the algorithm progresses and locations become scare, the tour can become very inefficient.

2) Modified Lin-Kernighan Heuristic: One of the most successful heuristic methods of generating near-optimal solutions to the TSP is the Lin-Kernighan Heuristic (LKH). In particular, a modified implementation by Keld Helsgaun has been shown to find the optimal solution for all tested problem instances with known optima [3]. Helsgaun’s algorithm is capable of optimizing tours with thousands of cities and runs in about \( O(n^{2.2}) \) time [3].

The success of Helsgaun’s algorithm comes at the cost of complexity and versatility. The implementation is more sophisticated than the original LKH, which itself is regarded as complicated. It is also designed only to handle symmetric travel costs. In order for the algorithm to be useful in a situation where there is an asymmetric augmentation of the travel cost graph, a transformation method must be applied to the graph to produce a symmetric one [3]. Jonker and Volgenant have designed such a method [4] (see also [5]), but it unfortunately doubles the size of the problem, in addition to adding implementation complexity. Most importantly, Helsgaun’s variation of LKH is not designed to account for arbitrary constraints and conditions. We know of no simple way to modify Helsgaun’s algorithm or LKH to account for generic, and potentially dynamic, costs that may arise in the computation of an optimal tour.

3) Genetic Algorithms: Evolutionary algorithms are a class of techniques for solving problems by mimicking concepts from biological evolution. Genetic algorithms (GAs) are a particular class of evolutionary algorithms that borrow the ideas of mutation and sexual recombination [6]. Genotypes are encoded in a structure chosen by the programmer to represent the information that may be included in a solution to a given problem. In some cases, mere bit strings will do, while in others, more complicated structures are necessary. A genetic algorithm works on a population of potential solutions, referred to as individuals. The population is usually random to begin with but can be initialized with another heuristic algorithm. After initialization, a stochastic search process is employed to iteratively improve the average fitness of the population. There are two common search strategies: steady-state and generational.

In a steady-state GA, the algorithm continues by picking a subset of the population according to some selection strategy. Members of the subset undergo recombination, with some probability. Usually the best individuals are chosen for recombination. The recombination technique depends on both the data representation for the genotype and any knowledge the programmer has of the problem space. Some recombination methods produce a single offspring, while others produce multiple. The offspring then undergo small random mutations, with some likelihood. These individuals are then inserted into the population, replacing existing individuals chosen according to some selection strategy. The process continues for many iterations or until some other terminating condition is met.

Generational algorithms work in much the same way as steady-state GAs. However, instead of continuously improving a single population, generational algorithms create new popu-
A steady-state genetic algorithm is used to find a near-optimal inspection ordering. The evolution parameters are summarized in Table I.

1) Data Representation: The algorithm creates a directed graph to represent travel distances between each pair of inspection points. Distances are calculated as simple Euclidean distances, but see Section III-B for a case where a more complicated calculation is used. Individuals are vectors of unsigned integers, representing indices into a vector of inspection points. An individual thereby represents a permutation of the inspection points.

2) Fitness Calculation: The fitness of an individual is calculated as the sum of the distances between each of the inspection points, starting with a point representing the AUV’s starting position. The cost to return to the AUV’s starting position from the last inspection point is not computed. This is a slight variation of traditional TSP.

3) Tournament Selection and Replacement: Four random individuals are chosen at the beginning of each iteration. The best two of these are chosen to participate in crossover and mutation. Tournament replacement is used to insert the resulting individual into the population. In this phase, the worst of four random individuals is replaced by the new individual.

4) Crossover and Mutation: Crossover has a 90% chance of occurring. When crossover is performed, order crossover is used to generate a single child from two parents. Order crossover was chosen because it has been shown to work well with permutations [7] and because it is easy to use with our data representation. If crossover is not performed, the better of the two parents is passed on for mutation.

Mutation always occurs. A randomly chosen portion of the permutation is chosen for inversion. Inversion mutation simply reverses the chosen section. It is worth noting that this operation is equivalent to a 2-opt algorithm, which LKH generalizes.

5) Terminating Condition: The algorithm stops running when any of a number of conditions is satisfied:
- Max runtime reached
- Max iterations reached
- Max iterations without improvement reached

When one of these conditions is encountered, the algorithm returns the best individual in the population. The runtime criterion is important for the intended application, since AUVs will be expected to make decisions and operate in real-time.

B. Currents as an Extensibility Example

One of the primary motivations for using a GA to solve the TSP is extensibility. Since we are interested in the practical usefulness of the algorithm, we must anticipate there being conditions which will affect the performance of the algorithm if not accounted for in the model. With the right pressures applied to the evolution, it is possible to accommodate these variations to the basic problem. For example, if a constant current will affect the trajectory of an AUV during an actual mission, it is necessary to simulate the current in the GA in order to generate efficient tours. Another example would be accounting for the minimum turning radius of the AUV platform. A large turning radius with multiple closely spaced targets becomes a significant factor in finding optimal tours [8]. By simply modifying the fitness function to take into account the true cost of a path, including currents, turning radii, or other real-world factors, it is easy to account for most real-world concerns. We have chosen to use currents to demonstrate the flexibility of the proposed GA.

1) Mathematics of Motion with Constant Current: Simulating a current breaks down to calculating the additional energy required to fight the current as well as the reduced energy required to travel with the current. Using some trigonometry, we can account for these effects on an AUV. The following derivation is adapted from [8]. Figure 1 illustrates the situation. The symbols used in the derivation are summarized below, in the order in which they are introduced:

\[
\begin{align*}
    u & \quad \text{AUV speed (m/s)} \\
    t & \quad \text{Actual travel time (seconds)} \\
    d_e & \quad \text{Effective travel distance (m)} \\
    d & \quad \text{Actual distance between MLOs (m)} \\
    u_r & \quad \text{Resultant AUV speed (m/s)} \\
    u_D & \quad \text{AUV speed in desired direction (m/s)} \\
    u_{CD} & \quad \text{Current speed in desired direction (m/s)} \\
    u_{CN} & \quad \text{Current speed 90 degrees to the left of the desired direction (m/s)} \\
    u_c & \quad \text{Current speed (m/s)}
\end{align*}
\]
The only unknown left is the component of the current speed in a direction perpendicular to the desired travel direction.

\[ u_{cN} = u_c \times \sin(\theta_c - \theta_D) \]

In order for the AUV to be able to travel from \( a \) to \( b \), the following requirements must be satisfied:

\[ |u_{cN}| < |u| \]
\[ u_r > 0 \]

The first requirement states that the speed of the current in a direction perpendicular to the AUV’s desired direction of travel must be less than the speed of the AUV. The second requirement states that the resultant AUV speed must be positive. If either of these conditions fail, the AUV cannot travel from \( a \) to \( b \), so an infinite travel cost is assumed.

2) Implementation: Extensions to the GA accommodating variations in the basic problem usually involve modifications to the fitness function. This is the approach that we could have taken for the case of constant currents. However, we are able to make a performance improvement by taking advantage of the fact that the GA does not rely on a symmetric cost graph. Since the travel costs between each pair of MLOs is constant, the directed graph representing these costs can be created once and remain cached throughout a complete optimization. Although some heuristics would not be able to optimize over the asymmetric data in the graph, the GA has no problems with it. We have implemented this as a routine that runs just before the GA. Therefore, this particular extension was possible without even modifying the existing optimization routines.

IV. INTEGRATION WITH MOOS

The inspection ordering genetic algorithm is integrated into the MOOS framework as a CMOOSApp module, plInspectionOrderer. The module subscribes to updates on the currently assigned waypoint locations, which are posted to the MOOSDB by a pre-existing behavior module. In our case, the waypoint locations represent MLOs. plnspectionOrderer also listens for updates to the current AUV position so that optimizations can take advantage of this information. When an updated list of locations is received, the algorithm is run until one of the terminating conditions is met. The maximum runtime and maximum number of iterations are configurable via a MOOS process configuration file for the module. Once the optimization is complete, the ordered list is published to the MOOSDB for other modules to use.

Since AUVs are expected to receive multiple waypoint updates, the plInspectionOrderer module is designed to use previous optimization results as a partial seed for successive optimizations. New waypoints are added to the end of a previously optimized tour to make a new tour, which are then optimized more quickly than a random tour. Also, if multiple waypoint updates are discovered since the last optimization was performed, only the most recent data is used.

V. EXPERIMENTAL SETUP

Genetic algorithms are stochastic processes. Therefore, a series of experiments are set up to determine the average effectiveness of the GA. Although the algorithm is designed for on-line operation, only the off-line results are shown here. Tests are performed under Ubuntu Linux with an Intel® Core™ 2 Duo T6500 running at 2.1GHz.

Since the algorithm in question must be implemented on actual AUVs, the experimental setup is designed to be reasonably realistic for the specific problem of mine detection and neutralization. Mines are generally dropped off surface ships as they travel in straight lines. To simulate this, ten mines are equally distributed in a straight line. Two parallel lines...
are created in this way. These lines represent actual mines, and we assume surface scanning equipment locates 100% of them, identifying them as MLOs to be inspected.

Surface detection equipment is not always accurate, however. There can be many false positives during initial scanning, generated by underwater debris. It is left up to the AUVs to determine which of the objects detected are actual mines, thus every object to be inspected is classified as an MLO. To simulate false positives, random inspection points are added to the search area. The percentage of MLOs which are false positives is adjusted between 0% and 70%. Thus the total number of MLOs for a distribution can be calculated as

\[
\text{Number of MLOs} = \frac{20 \text{ mines}}{1 - \text{false positive rate}}
\]

For each distribution, a single inspection map is used in all trials for repeatability. An example mine distribution as seen within the MOOS simulator is shown in Figure 2.

The AUV starting location is the same in all trials. It is placed outside the search area, since the surface ships delivering the AUV will need to stay in safe waters. The starting position is marked in Figure 2.

VI. RESULTS

A. Basic Results

In order for the GA to be a viable approach to creating an efficient inspection ordering, it must be measurably better than simpler approaches. More specifically, it must consistently produce shorter tours in an acceptable amount of time. Table II compares the average results of the GA over 25 trials versus the nearest-neighbor heuristic. The GA produces shorter tours than NN in every non-trivial case. The small variance with the GA also indicates consistency, which is important for verifying that the GA has not merely gotten lucky.

Figure 3a shows the results of applying NN to the largest test case. Figure 3b shows a typical tour generated by the GA on the same data set. The simplicity of the tour generated by the GA compared to that of the NN provides visual confirmation that the GA performs better on the test data. In particular, one can see that the early part of the tour generated by NN appears reasonable, while the latter part has more crossings and is less efficient. This behavior does not apply to the GA, as it optimizes over complete tours.

The GA also runs in an acceptable amount of time on the test hardware. The maximum iterations without improvement criteria, set to 500k iterations, was the terminating condition met in all our experiments. Results for the largest test case were typically returned in under 10 seconds, and the GA typically converged on these optimizations in less than 5 seconds. Although runtime scales positively with the number of MLOs, we did not examine the exact relationship. In particular, examining the performance on larger problem instances is not reasonable, given the limited battery resources available on modern AUVs. Figure 4 shows a plot of the tour cost versus time for several runs, as applied to the largest test case. The graph shows the results of the nearest-neighbor algorithm as a horizontal line for comparison. This shows us that the GA outperforms NN very quickly.

B. Current-Aware Algorithm Results

Using the above formulas, we were able to pre-compute the effective distance that an AUV must travel between each pair of MLOs. These distances replaced the distances in the GA's directed graph. No further modifications were necessary to optimize over the new data sets. Table III compares the effectiveness of the GA and NN techniques when accounting for constant currents. Again, 25 runs of the GA are performed for each MLO distribution. Notice that the GA performs better than NN in all trials.

VII. CONCLUSIONS

The application of a genetic algorithm to the problem of ordering the inspection of MLOs for autonomous vehicles has proven to be effective in reducing the cost of a tour relative to the provided NN behavior. The average improvement for non-trivial cases is over 14%. Also, the approach is extensible and allows for the application of additional behaviors such as compensating for ocean currents or turning radius limitations.

ACKNOWLEDGMENT

The authors acknowledge support from the Office of Naval Research under the award “Cooperative Autonomous Underwater Vehicles used to Search Large Ocean Areas for Mines”, No. N00014-08-1-0276.

REFERENCES

TABLE II

<table>
<thead>
<tr>
<th>Number of MLOs</th>
<th>Noise</th>
<th>Nearest Neighbor</th>
<th>GA, 25 Trials</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Tour Length (meters)</td>
<td>Average Tour Length (m)</td>
</tr>
<tr>
<td>20</td>
<td>0%</td>
<td>4785</td>
<td>4785</td>
</tr>
<tr>
<td>25</td>
<td>20%</td>
<td>5985</td>
<td>5573</td>
</tr>
<tr>
<td>33</td>
<td>40%</td>
<td>10024</td>
<td>8028</td>
</tr>
<tr>
<td>50</td>
<td>60%</td>
<td>11752</td>
<td>10225</td>
</tr>
<tr>
<td>67</td>
<td>70%</td>
<td>14562</td>
<td>1190</td>
</tr>
</tbody>
</table>

(a) Nearest neighbor path
(b) Typical GA optimized path

Fig. 3. Optimizations for 67 MLOs.

TABLE III

<table>
<thead>
<tr>
<th>Number of MLOs</th>
<th>Noise</th>
<th>Nearest Neighbor</th>
<th>GA, 25 Trials</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Tour Length (meters)</td>
<td>Average Tour Length (m)</td>
</tr>
<tr>
<td>20</td>
<td>0%</td>
<td>9496</td>
<td>9309</td>
</tr>
<tr>
<td>25</td>
<td>20%</td>
<td>12322</td>
<td>9895</td>
</tr>
<tr>
<td>33</td>
<td>40%</td>
<td>18836</td>
<td>17703</td>
</tr>
<tr>
<td>50</td>
<td>60%</td>
<td>27346</td>
<td>22050</td>
</tr>
<tr>
<td>67</td>
<td>70%</td>
<td>29086</td>
<td>25079</td>
</tr>
</tbody>
</table>


Fig. 4. Convergence speed of the GA for 67 MLOs. Four typical runs of the GA are shown. A horizontal line showing the result of the nearest-neighbor algorithm is added for comparison.