Learning Area Coverage for a Self-Sufficient Hexapod Robot Using a Cyclic Genetic Algorithm

Gary Parker, *Senior Member, IEEE,* and Richard Zbeda

***Abstract*—Self-sufficient autonomous robots are able to perform independent tasks while maintaining enough energy to function. We develop a self-sufficient robot control system where a cyclic genetic algorithm (GA) is used to learn the control program for a hexapod robot equipped with a quick charge power supply. This robot uses high capacitance capacitors for its onboard power storage and a sensor system to detect power need related information. A detailed simulation is developed, to be used by a cyclic GA to learn control programs for the robot. These programs enable it to perform area coverage and periodically return to a recharging station to maintain power. In this paper, we expound on previous research and report the transfer of the complete simulated self-sufficient behavior to the physical robot and colony power supply system, where tests have been conducted to confirm the viability of our approach.**

***Index Terms*—Colony robots, control, evolutionary robotics, genetic algorithms, self-sufficiency.**

I. Introduction

ROBOTS in a colony must be capable of performing independent and long-term colony tasks within their respective environmental systems. This requires two qualities: autonomy and self-sufficiency [1]. The robots must make their own choices and manage their own activities independently, and they must perform in a system where they can operate for long periods of time by some means of recharging their own power storage. To have self-sufficiency, the robot would need to either have an onboard energy generator or have a means of recharging an onboard power supply via a charging station. In the latter case, as opted for this paper, the robot must also have a means of examining its own power storage to know when to recharge and must be able to locate and use a charging station.

Implementations of a self-sufficient power supply have included solar powered batteries [2] and batteries that recharge at a power station [3] as described by McFarland and Speir, who specified the basic cycle of an autonomous and self-sufficient robot as *work—find fuel—refuel*. Wawerla and Vaughan [4] consider recharging verses work strategies for autonomous robots, offering a model of when the robot should recharge verses work. In our research, we are allowing the genetic algorithm (GA) to learn when to recharge verses work. In the previous paper, we presented the use of capacitors for a rechargeable power supply [5]. Capacitors recharge and consume their charge faster than batteries. This allows for a more rapid change between work and refueling, which is effective in situations where continual robot activity is needed. An additional benefit of capacitors is that they are much more resistant to aging side effects than batteries. As rechargeable batteries are used over time, they gradually lose their ability to hold the same levels of charge consistently [6]. These limitations make them inappropriate for long term tasks where there is little or no opportunity to change batteries.

The research reported in this paper deals with the unique challenges of learning control for a hexapod robot equipped with capacitors—it addresses the needs of the entire robot system. The task that the robot was assigned is area coverage, where the robot must cover as much area as possible. However, while performing its task, the robot needs to keep track of its power storage and navigate back to the recharging station to recharge when power is low. The robot's power sensor was configured to monitor the robot's power status effectively and the robot was configured with sensors that allowed it to sense the location of the charging station [7]. The return (finding its way back to the charging station) and area coverage control tasks were incrementally learned using a cyclic genetic algorithm (CGA) in a simulation [8], [9]. The use of a learning system to develop control programs can reduce the initial work of the programmer and adds the possibility for adapting to changes in the robot's capabilities and the environment. The CGA was chosen for this research because of its ability to learn multiloop control programs that can be directly downloaded to the robot. In addition, this research extends the use of a CGA to this particular problem. In the research reported in this paper, feedback from tests of the learned behaviors on the actual robot was used to make improvements in the simulation input details, such that the control programs produced improved in effectiveness. The improved control programs were tested on the actual robot to verify the viability of the system.

A. Evolutionary Robotics

Robot behavior can be developed by preprogramming or learning. Preprogramming the robot controller may produce

Manuscript received September 13, 2011; revised September 7, 2012; accepted September 14, 2012. Date of publication January 9, 2013; date of current version August 21, 2014.

G. Parker is with the Computer Science Department, Connecticut College, New London, CT 06320 USA (e-mail: parker@conncoll.edu).

R. Zbeda is with Premier Technology Solutions, New York, NY 10016 USA (e-mail: rzbeda@gmail.com).

Color versions of one or more of the figures in this paper are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/JSYST.2012.2223071

1932-8184 © 2013 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission.
See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

the desired behavior, but is laborious for the developer. In addition, it does not provide the ability of the robot to adapt to circumstances not observed or specified in the preprogrammed behavior, such as miscalculation, a changing environment, or degradation of the robot's mechanical system. On the other hand, learning behavior through some means of evolutionary computation (EC) can obviate laborious development and have the potential to provide adaptability [10].

Past research has used EC to learn artificial neural network weights and fuzzy systems, genetic programming, and CGAs to learn robot control programs. In particular, using GAs to learn the connection weights/architectures for artificial neural networks has been common [11]. Beer and Gallagher [12] used GAs to successfully evolve neural networks for chemo-taxis and legged locomotion controllers. Floreanno and Mondada [13] developed the Khepera robot and evolved neural networks to control navigation and obstacle avoidance behavior on one that had to periodically locate a charging station before its simulated batteries lost power. Lund and Miglino [14] developed wall and obstacle avoidance behavior for a Khepera robot through neural network controller evolution. Zhou and Er [15] used evolutionary algorithms to learn fuzzy control systems for a Khepera robot.

Genetic programming (GP) is also a commonly used option. Busch *et al.* [16] used GP to evolve robot controllers for gait production for simulated robots independent of their specific morphology. Lazarus and Hu [17] used GP in simulation to integrate controller development with sensor input for simulated robots performing obstacle avoidance and wall-following tasks. Nordin *et al.* [18] also use GP to evolve wall-following behavior for simulated robots.

The cyclic GA is unique to these methods of evolutionary computation because it is designed to learn the code that can be directly downloaded to the robot controller. In other works, it is a method of automatic code generation that can be used on the low-level programming language of the robot.

B. Area Coverage

In previous work, we used computational intelligence to learn in simulation the two phases of self-sufficient behavior: area coverage for when the robot is working and return for when the robot must find fuel [8], [9]. We separated the two tasks and incrementally evolved them. Incremental evolution was proposed by de Garis [19] and is a commonly used approach in evolutionary robotics [20]. The return task was first learned and then became a predefined modular behavior that was usable when learning the area coverage task. This approach was chosen because the effectiveness of the area coverage behavior was highly dependent on the effectiveness of the return behavior.

Evolving control behavior for a robot to locate and travel to a target has been addressed in numerous studies. Tuci *et al.* used an artificial neural network (with a GA for the learning) to direct a Khepera robot to navigate toward/search for a target placed at one end of the arena [21]. Bajaj and Ang [22] analyzed controller evolution using an incremental approach on a Khepera robot doing navigation and obstacle avoidance. A neural controller was used to direct straight navigation while avoiding obstacles and later extended to wall following behavior. Ram *et al.* [23] used GAs to learn navigation for robotic reactive control systems in dynamic environments, although only simulation results were obtained. Georgescu and Parker [24] evolved a multiloop controller with a cyclic GA for a wheeled LEGO robot with sensors. The task chosen was navigation through an obstacle maze toward a light source.

Incrementally learning the area coverage task over navigation behavior has not been a widely addressed topic of research. Previous research in the area of coverage path planning initially concentrated on preprogramming robot behavior to cover a specified area while contending with obstacle avoidance. In addition, controllers for autonomous agents performing area coverage have been created using neural network [25] and constrained motion models [26]. Parker reports several methods used in previous publications [27], [28]. However, common to these initial area coverage methods is either precise control of the robot or some navigational means of finding its position/orientation and making continual corrections to its movement. These assumptions make them ineffective for legged robots with minimal sensors and precision of movement since they cannot be positioned perfectly with exact headings. In particular, a capability to perform perfect back-and-forth boustrophedic motions [29] is especially difficult for legged robots since the exact time and rate of turn cannot be specified. If the robot cannot make tight turns very accurately, it may be more efficient for it to make overlapping sweeps of the area instead of following the boustrophedic pattern.

To address these issues, methods have been used for coping with the inaccuracies in legged locomotion. For example, Reinstein and Hoffmann [30] used an inertial navigation system along with legged odometers. Learning methods have also been used for adaptation to specific capabilities in legged robots performing area coverage. Parker [27], [28] developed a method for learning turn cycles for a hexapod robot with a CGA that produced the tracks required for area coverage. The CGA allowed the system to learn area coverage despite the inaccuracies of the legged robot, yet did not require extensive upgrades to the robot such as inertial navigation and odometers on the legs.

Our approach consists of three different parts. Firstly, we utilized a continuous power supply system in the form of capacitors as an onboard power supply for hexapod robots, and an external power station used for recharging the capacitors [7]. This power supply system was integrated such that the hexapod robot can find fuel and recharge through the use of several sensors. Secondly, the dual tasks, area coverage and return, are incrementally learned offline in simulation with a CGA [8], [9]. Thirdly, the behavior learned in the simulation is transferred to the physical colony robot, such that it implements the learned behavior in the physical colony space and with the actual power supply system. General observations are made of the physical robot and then used to improve the simulated learning.

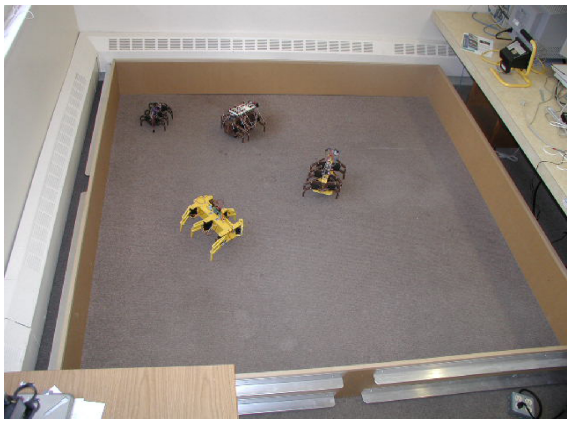

Fig. 1. Colony space in the Connecticut College Robotics Laboratory.

II. Robot and Power Supply System

The colony space is an 8 × 8 ft walled area in the Connecticut College Robotics Lab (Fig. 1). Our autonomous and self-sufficient robot approach exists within this experimental framework.

A. ServoBot

The robot used was the ServoBot, which was developed for legged robot and colony robotics experimentation by David Braun at Indiana University. The ServoBot is an inexpensive hexapod robot built out of Masonite wood. Each of its six legs has two servos to power horizontal thrust and vertical movement thus enabling two degrees of freedom per leg. The ServoBot as originally configured carries onboard one 9 V battery and 4 AA batteries to function as its power supply [31]. The 9 V battery powers the onboard BASIC Stamp II microcontroller and the 4 AA batteries power the servos. The BASIC Stamp II has pins used to send control pulses to the servos. It stores and executes a sequence of servo control pulses in the control program. The pulses sent out by the control program command all 12 servos to move to their next specified location. A proper sequence of these activations produces a gait cycle, defined as where the robot's legs begin moving and eventually return to their original position after following a full step cycle of positions. The servos require 5 V to operate and 12 servos running at the same time can be powered by four fully charged AA batteries.

One of the advantages of a learning system is that the researcher does not need to know all of the kinematics and dynamics of what is going on with the robot, he/she just needs to know the results of each control input. Nevertheless, some explanation of the leg design is presented here for those who are interested. Fig. 2 shows the front of a typical ServoBot. Each leg is made up of six parts. Part A is connected by a hinge to the center line of the robot. It is approximately 80 mm across the top, 80 mm on the hinge side, and 60 mm on the other side. This part has a servo mounted in its center with either a disk or arm attached to the servo's shaft. This disk (or arm) transfers the rotary movement of the servo into linear motion of an extension arm attached to the disk.

The disk can be controlled to move 180° with it set so that the extension arm connection is either all the way up or all the way down. In the photo (Fig. 2), it is all the way up and a counterclockwise motion will move it all the way down. Since the extension arm is connected to Part C (80 mm in length), this motion will pull down on C a distance equal to the diameter of the disk at the connection point. Part C is connected to A at one point, which can be seen just down from the pointer associated with C. Part D (60 mm in length) is also connected to A, and B (110 mm in length) is connected to both C and D. The movement of C down at the servo connection point moves B up. With the connections of A, B, C, and D creating an approximate parallelogram, B stays vertical, with only slight in and out movement, as it moves up and down. The length of C is about 100 mm with the distance from the servo connection to the Part A connection in the 38 to 42 mm range and the Part A to Part B connection in the 44 to 48 mm range so the movement of the leg up and down is somewhat more than the diameter of the servo disk. The labels E and F are identifying parts that have the same length as Part D. These are mounted on the back side of Parts C and D to aid in stability. A servo mounted on the base of the robot is connected to the bottom of Part A in the 72 to 76 mm range from the hinge. This allows the leg to move back and forth as the servo rotates.

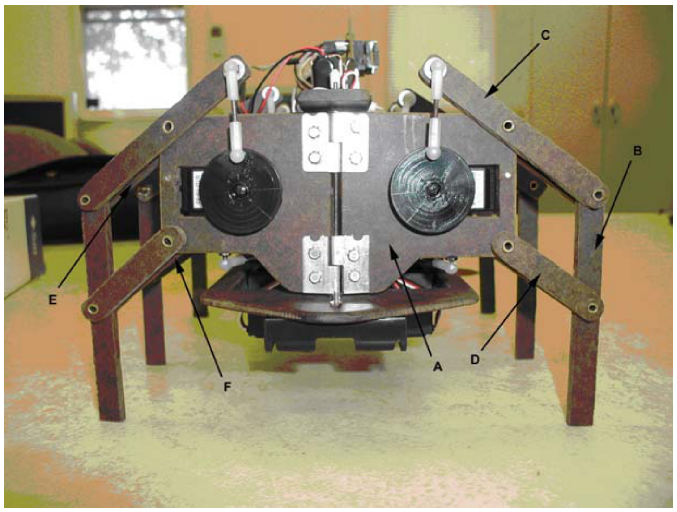

Fig. 2. Front view of the ServoBot, showing the parts making up the legs.

The pulses sent out by the BASIC Stamp to these two servos drive the servos all the way up or down in the case of the servo mounted on the leg, or all the way back and forward in the case of the servo mounted on the base. A proper sequence of these pulses results in a reasonable leg cycle and a proper sequence of pulses to all of the servos in parallel results in a gait. With these control pulses required every 25 milliseconds, the normal gait for this robot consists of a sequence of 29 servo pulses to move the leg from the full front to the full back position. Since anything less than 29 will result in some dragging of the legs on that side, this can be used to form varying degrees of turn.

B. Power Supply System

It was determined in previous research [5] that, although the 9 V batteries could provide sufficient control power, the 1.5 V batteries for motor power were inappropriate for lengthy hexapod robot experiments. Our solution was to use capacitors, specifically, Panasonic Electric Double Layer Radial Lead

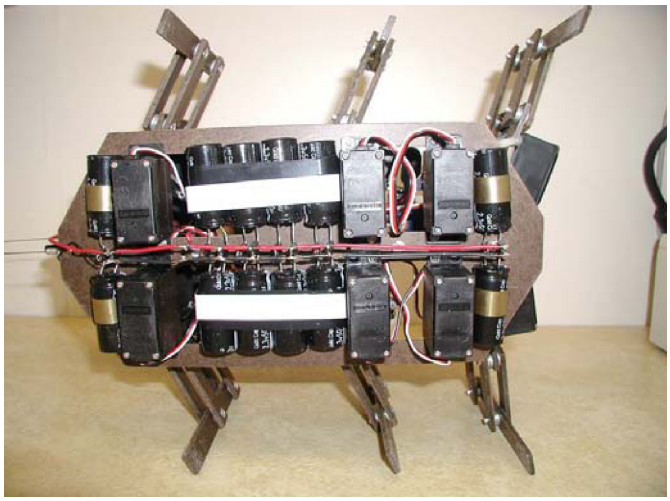

Fig. 3. ServoBot's six pairs of capacitors.

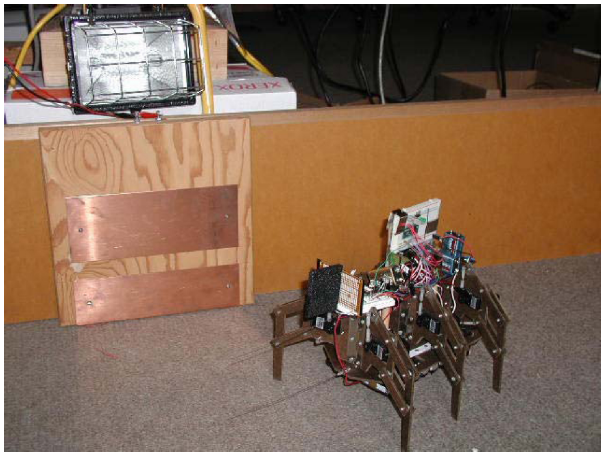

Fig. 4. Robot and its metallic probes, power station with its metal plates, and light source.

Capacitors (Gold Cap). They have a 50 F capacitance and are rated for a maximum potential difference across the leads of 2.3 V. There were six parallel groups of two capacitors connected in series for a total of 12 capacitors. Each parallel group provided a 4.6 V level of charge to power the servos (rated for 5 V). Six pairs of these capacitors were connected in parallel in order to provide a greater capacitance for increased power storage [5]. The aggregate capacitance of capacitors in parallel groups is the sum of their capacitances. The total capacitance of the system is 150 F with six pairs of 50 F capacitors. Such an arrangement meant that the power available included a total capacitance of 150 F and a 4.6 V level of charge. The capacitors were placed underneath the robot since this design provided more stability to the robot, was more compact, and left the top of the robot available for other items such as sensors. Fig. 3 shows a photograph of the underside of the robot where the capacitors were mounted.

The capacitors would power the servomotors and would be recharged when the robot made contact with the power station. Recharging was accomplished through the robot's two frontal metallic probes, which connect the robot's capacitors to the metal plates of the charging station. The capacitors have "polarity," so the negative pole of the power station needs to be connected to the negative lead on the capacitor; the same needs to be done for the positive pole. The robot, the power station and the metallic probes are pictured in Fig. 4.

The charge time was decreased by setting the power supply voltage level higher than the capacitors maximum accepted voltage and stopping the charging when the maximum accepted voltage was reached. This was particularly effective since the graph of voltage versus time for charging a capacitor is an exponential graph. Tests were done to determine the run and charge times when the capacitors were charged at 4.6 V for our specific capacitor configuration. With this system, the ServoBot had a charge time of 2 min 20 s and had a run time of 2 min 50 s [5].

C. Sensors

This research involved devising a control scheme that allowed the ServoBot to effectively make use of an adjusted version of our previously devised capacitor centered power supply system reported earlier [7]. The robot was equipped with two object detection sensors (IROD) on the front to enable wall avoidance; a voltage sensor to tell that it was time to charge, that it was charging, and that charging was complete; and two light sensors to detect a source placed over the power station. These sensors were coordinated by onboard BASIC Stamp II controllers.

A PIC 12F675 [32] microcontroller was used as an analog-to-digital voltage converter (ADC) to sense the voltage level of the capacitors. It was configured to detect when the power level was greater than 5.6 V (indication to stop charging), 3.8–5.6 V (normal operations), less than 3.8 V (indication that it was time to seek power), and less than 3.4 V and a slight jump in voltage was detected (indicated contact between the robot's metal probes and the power station's metal plates).

Two CdS (cadmium sulfide) photocells were used as light sensors, and a BASIC Stamp II was used to process the light sensor input. These light sensors allowed the robot to detect the light source using a method that identifies four states through the use of empirically defined thresholds and represents them in two bits. The four states are as follows: when both cells are in a shadow, when both cells are in full light, and when one cell is in shadow and the other is in full light.

Two infrared object detection (IROD) sensors (Sharp GP2D05 IR Rangers) were used for wall avoidance. They were positioned in front of the robot with a span of 45° and situated so that their spans overlap by 10°. In this way, the robot can detect whether objects (i.e., walls) are directly in front (in which both sensors sense an object) or at either of the frontal sides. The threshold ranges were adjusted to detect objects within 39 cm from the center of the robot.

D. Controller

The control scheme integrated four BASIC Stamp II's controllers, including a walking, main, IROD sensor controller, and light sensor controller each with a 9 V battery. It also involved a PIC Chip voltage sensor (ADC) controller, which is powered by a BASIC Stamp II regulated 5 V out pin.

The system diagram is shown in Fig. 5. The main controller,

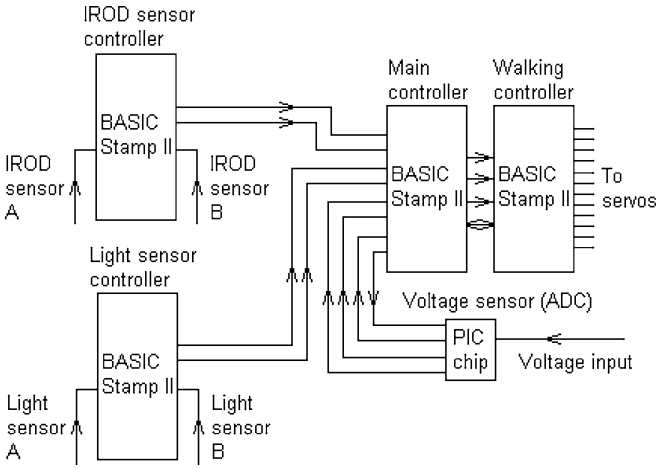

Fig. 5. Wiring scheme with main controller, walking controller, voltage sensor (ADC), IROD sensor controller, and light sensor controller.

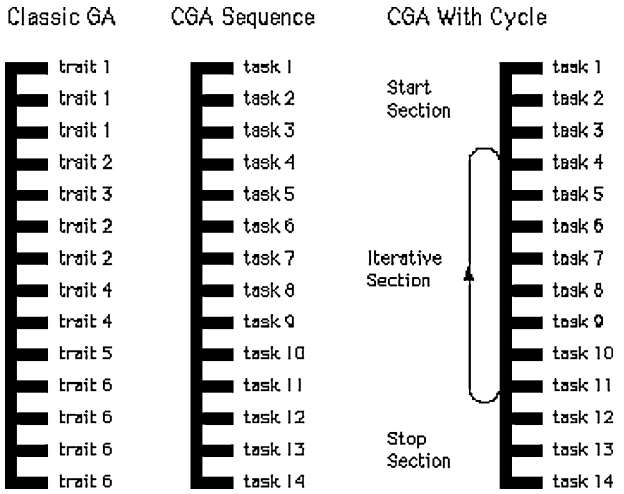

Fig. 6. GA and CGA chromosomes.

in its execution of the learned control program, would take sensor data into account before executing a branch instruction. When executing a move instruction, the main controller would send 4 bits of information to the walking controller, indicated the 1 of 16 types of gait cycles for the walking controller to coordinate and execute.

III. Cyclic Genetic Algorithm (CGA)

A GA [33] uses survival of the fittest and heredity to learn solutions to problems that are normally too computationally intensive for standard brute strength algorithms to solve. They use a population of chromosomes made up of 0s and 1s to represent the solution to the problem. The initial population is randomly generated at the start of learning. At each generation, a new population is formed using selection, crossover, and mutation. After all the individuals in the population are tested for fitness, two are selected to be the parents. They recombine using crossover—the offspring is a combination of their chromosomes. After this, the new offspring is subjected to a very low probability of mutation where one or more of its bits may be flipped (0 to 1 or vice versa). The selection/crossover/mutation process is repeated enough times to create a new population of offspring. After several generations, the intent is that one of the individuals in the population will be a near-optimal solution.

The CGA [34], a variant of Holland's GA [33], was created to generate code for cyclic control problems. Whereas the genes of a GA correspond to traits of a solution, the genes in a CGA chromosome correspond to tasks to be completed. A CGA chromosome is a list of instructions that incorporates loops to execute a sequence of repetitive tasks (as shown in Fig. 6). There can be loops of single instructions, loops over other loops themselves, and loops over the whole chromosome.

CGAs are appropriate to use to learn self-sufficient behavior because of the cyclic nature of the *work-find fuel-refuel* cycle. Initially, the CGA was used in evolving single-loop control programs for hexapod robot gait cycle execution [31], [35]. The main part of the chromosome was composed of a single loop to address one repetitive overall task. However, a single loop was inadequate for general robot control programs with the need for different subloops to respond to dynamic sensor input.

A CGA with conditional branching [35] was created to facilitate branching to different loops in response to specific sensor inputs. The initial problem addressed was the development of a search program for a predator robot to find a stationary prey. The 128-b chromosome, which was designed to accommodate four different states, had four segments, each of which represented a control loop or a cycle that the robot repeated as long as the sensors' inputs stayed the same. The segments were fully connected, so control could jump from any segment to any other segment. Each segment consisted of four genes and each gene consisted of a pair of integers. The first integer of the gene determined which action was to be taken and the other dictated the number of repetitions of that action. After the robot performed the current action the specified number of repetitions, it would check the state of the sensors. If the sensor states remained the same, the robot went on with the next gene in the same segment. After the execution of the last gene in the segment, the cycle continued at the beginning with the first gene in the segment. If at any time the sensor inputs changed between genes, control jumped to the first gene of the segment that corresponded to the new sensor inputs. This worked well for the problem being solved, but it required a segment for every possible combination of sensor inputs and the multiloop program did not work with continuous sensor values.

The CGA with conditional branching method was further modified in order to deal with a large number of sensors [24]. In this paper, the control program was allowed to have up to seven loops. It had seven parts to each loop. The loops had a 2-b number, which designated whether it was to loop once, twice, three times, or infinitely. Each loop had 6 instructions. Each instruction was either a control instruction for the robot or a jump instruction. A jump would take place whenever the sensor combination comparison designated in the bits of the instruction was true and the jump location was to the beginning

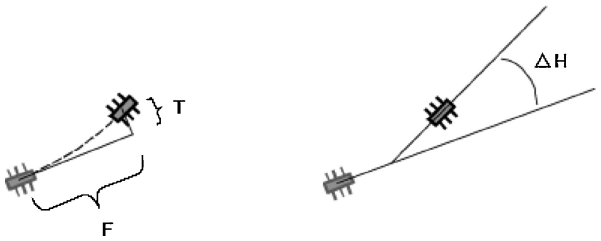

Fig. 7. Gait cycle turn measurements. F and T are shown in the left diagram. F is the distance moved forward from the start position. T is the distance moved perpendicular to the start position heading. ΔH, which is the change in heading, is shown in the right diagram.

of any of the seven loops. With this change, the CGA was now capable of learning when to execute sensor branching tests, which sensors to test, and what chromosome segments to jump to.

Such a multiloop CGA with conditional branching was used to learn the control program for a simulated hexapod robot doing area coverage (i.e., work) and return (i.e., finding fuel). The simulation simulated the colony space, robot, and self-sufficient power supply system. The resulting evolved behavior was designed for feasible transferability to the physical system.

IV. Simulation

The simulation was modeled after the ServoBot and its gait cycles [8], [9]. In this context, a gait cycle is a full step cycle where each leg completes all the positions in a step and returns to its starting position. The physical robot's standard gait is a control sequence or list of activations that the onboard controller continuously sends. Activations provide the control for the 12 servo actuators' instantaneous movements. In previous work a CGA was used to evolve an optimal gait for the ServoBot [31]. The resulting tripod gait is an optimal gait for speed. Affecters were used to produce differing degrees of turn in the gait cycle. They were set on a scale from 0 to 5 on either the left or right side (but not both), meaning that there were a total of 12 possible affecter values used and as a result, 12 different possible turning gait cycle types.

Four more gait cycles were created to allow left rotate, right rotate, no movement, and straight backwards. With these four gaits, a total of 16 gaits were used. A rotate gait is a full turn with the nonturning side's middle leg doing a backwards thrust to increase the turn angle. A no movement gait is when the robot does not move its legs for a standard gait cycle's time duration. A straight backward gait is a no turn gait in reverse. Each of the gait cycles were measured for displacement and rate of turn (Fig. 7) and the results stored in a table.

The simulated robot moves and operates in the simulation area (Fig. 8) which is a 500×500 unit square walled area that represents the actual Connecticut College Robotics Lab 8×8 ft colony space. The simulated power station, which represents the physical power station, is centered on the right wall's center coordinate of (500, 250) and covers the 225–275 y range. The robot uses the power station by making contact with it. Its wires, which extend out 45 units in front of it, must make contact with the power station while the robot is angled

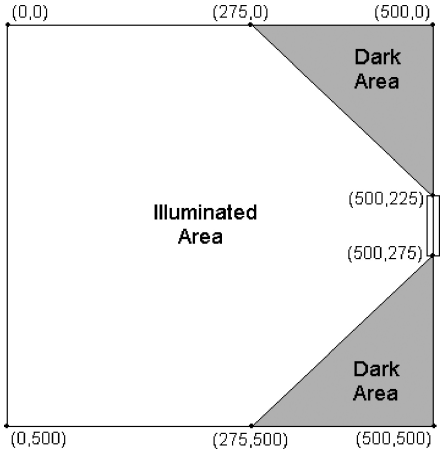

Fig. 8. Simulated colony space.

between 45° and 135° to connect with the charging station. The specific angle requirement was implemented because the probability of good contact reduces as the angle of the physical robot's probes in relationship to the plates gets further from 90°. In addition, the higher angles of contact require the side of the robot to be too close to the wall. If the robot's coordinates get within 25 units of a wall, the simulation run halts to simulate a collision with the wall. The simulated power station has a light source that emanates from two point coordinates [(500, 240) (500, 260)] and only affects a certain area of the simulated colony space, as shown in Fig. 8. Due to the positioning of the light and placement of the sensors, the robot can only sense light from a location in the illuminated area. In this way, the physical light source/sensor constraints are represented in the simulation.

The simulated robot, like the actual robot, has right and left wall detection sensors, right and left light sensors, and a power sensor. Power usage per gait cycle was measured and used to simulate power loss as the robot operated. If the voltage dropped below 2.7 V, the simulated robot stopped because it did not have sufficient power to move.

V. Learning the Return Module

A multiloop CGA with conditional branching was used to learn the return task, specifically the type that learns jumping between loops, which was created to handle many sensor input combinations [24]. The chromosome was designed with a sufficient number of loops to cover each subtask. It was also designed with a specific number of instructions in each loop to create enough conditional branch tests and/or gait cycle movements to make the loop function effective. However, the number of loops and loop instructions were not so large that the CGA did not converge on a good solution.

Each chromosome is made up of eight genes, and each gene is made up of a 2-b number followed by four 7-b numbers. The four 7-b numbers represented the instructions, one type being a specific gait cycle instruction, and the other being a conditional branching test instruction. The gene represents a "for" loop with the 2-b number indicating the number of repetitions of the loop's instructions where the possible values

```
0-no sensor
1-front-right object detection sensor
2-front-left object detection sensor
3-both object detection sensors
4-low power sensor
5-front-right light detection sensor
6-front-left light detection sensor
7-both light detection sensors
```

Fig. 9. Eight sensor test combinations.

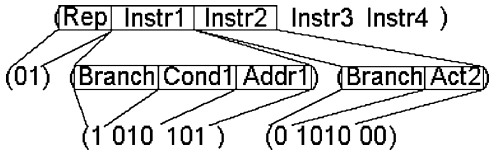

Fig. 10. Two different structure types of genes.

are 01 (one), 10 (two), 11 (three), and 00 (infinity). For each 7-bit instruction, the first bit indicates either a conditional branching test or a gait cycle to be executed. If equal to 0, the next 4 bits represent the one of 15 gait cycles to be executed and the last two bits are ignored. If equal to 1, the next 3 bits represents one of 8 predefined sensor input combinations. The 8 sensor combinations are listed in Fig. 9. The 3 bits after the conditional represent the address of one of the 8 gene segments to jump to if the actual sensor states match the conditional test. An example of a gene and its structure is shown in Fig. 10.

The flow of the fitness evaluation of a chromosome began with the first gene's list of instructions. After each instruction is executed the next instruction in the loop is processed unless a conditional branching instruction forces a jump to another gene. After a gene is finished executing, control transfers to the beginning of the next gene. If the last gene finished executing, control would return to the beginning of the first gene. This flow would continue until the run terminated. After 32 consecutive branches without a gait cycle instruction, the simulation was set to halt to avoid infinite branching with no activations.

For training, five populations of 256 randomly generated chromosomes were generated. 1024 generations of evolution were carried out. For each generation each chromosome was tested on the same 10 randomly generated starting positions. Each individual's fitness was set to its average performance in the 10 runs. To amplify fitness differences between individuals in a population this average was raised to the 1.5 power. After each of the chromosomes was assigned a fitness value, the chromosome with the best fitness was automatically included in the next generation. The remaining chromosomes to be populated into the next generation were produced by applying the three standard genetic operators: selection, crossover and mutation. At periodic generations from 0 up to 1024 throughout the evolution the populations were saved.

A fitness value was assigned to a single chromosome for a single run using the following procedure. A run would start with the robot's power level at the low power threshold and a low maximum of 200 steps to encourage it to travel directly toward the power station. Maximum fitness would be achieved by connecting with the charging station. If the robot got within

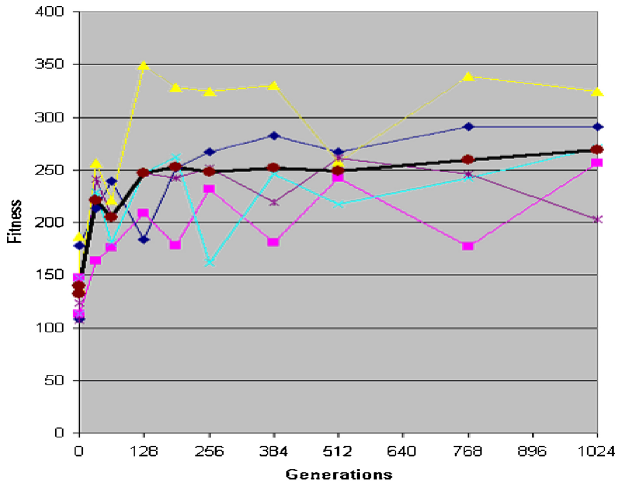

Fig. 11. Fitness of the power seeking module. For each population the best individuals at each generation were recorded (colored lines). The black bold line represents the average of the five tests.

25 units of a wall the run halted and a 0 fitness value was assigned. For all other cases, the closer the robot was to the power station when the run terminated, the higher the fitness.

Run termination and the fitness value assignment occurred for any of four reasons: the robot reached the power station, 200 steps (e.g. gait cycles) were taken; the robot got within 25 units of a wall (wall collision), or the robot's power level dropped below the empty power threshold.

A. Results

For each of the five tests (one for each randomly generated population), analysis was done on the population details for 0, 1, 32, 64, 128, 256, 384, 512, 768, and 1024 generations. In a particular test, the populations at each of these generations were evaluated from 100 randomly generated start positions. Results of the best individual fitness (averaged over the 100 trials) from each population at each stored generation were recorded (Fig. 11).

Observations of the simulated robot revealed that the individual with in best fitness in the 1024th generation out of all five populations would act in the following way. If the robot's sensors did not sense any light it would do a few turns. This would continue until the robot ran out of steps if it was in one of the dark areas. Otherwise, the robot sensed light, it traveled to the power station and would switch between two behaviors; while its right sensor sensed light it would make minimal right turns until its right light sensor did not sense any light; once this happened it would make one or a few medium left turns after which the first behavior resumed. This cyclic behavior was repeated until the robot reached the power station.

The results of all five runs of the return module show that the population fitness average reached a high point after a few hundred generations beyond which it modestly improved to a maximum at 1024 generations. The way in which the simulated robot improved behaviorally was through taking increasingly direct and immediate routes to the power station. Overall, using a multiloop CGA with conditional branching was shown to be an effective method for learning the return

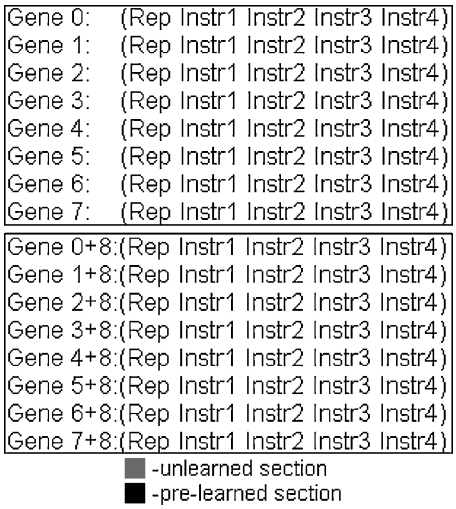

Fig. 12. Unlearned (area coverage) and a prelearned (return) section of the initial complete chromosome structure. Control was maintained in one section at a time and would alternate between the two.

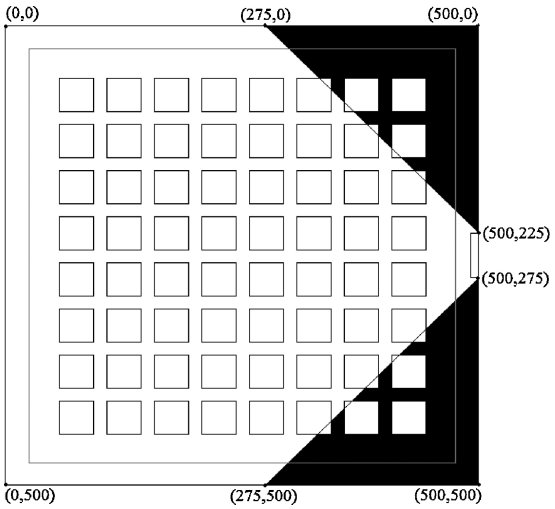

Fig. 13. Simulated colony area with the 64 square areas to be covered, the kill zone, the light distribution areas, and the power station. The figure is drawn to scale.

task in our simulation setup.

VI. Learning Area Coverage Using the Return Module

The area coverage behavior for the robot was developed incrementally from the return behavior using a multiloop CGA. Simulated robot behavior was evolved that would enable it to execute conditional branching tests to direct it to switch into the return mode when low on energy, and to do area coverage starting from the recharging area. The CGA was needed because different overall tasks such as leaving the charging station after refueling, covering wide sweeps over new colony area, avoiding walls, and rotating to position for additional wide sweeps over new area were needed. For the different tasks, different CGA loops were required as well as the right jumping between such loops. The robot had to learn to jump into return/fuelling mode when low on energy.

The area coverage chromosome structure was identical to the return chromosome structure. The 8 gene area coverage chromosome was combined with the 8 gene return chromosome to form a 16 gene chromosome (see Fig. 12). The 8 gene return chromosome used was the chromosome with the best fitness in the 1024th generation out of the five populations in the return simulation. This same prelearned chromosome was used for each 16 gene chromosome of each population in the area coverage simulation and did not change or evolve.

The flow of fitness evaluation was as follows. To induce the robot to learn to conduct a conditional branching test to test its low power sensor when low on power, the robot would start with a power level set at the low power threshold. When tested positive, the robot switched into the first gene of the return chromosome section and travel to the power station. After a charging routine the control automatically switches to the first gene of the of the area coverage chromosome. Once in area coverage mode, the robot executed its area coverage strategy through a series of straights and turns. Periodic conditional branching tests of the low power sensor allowed the robot to effectively switch back into return mode when low on energy and reach the power station to refuel to do more area coverage.

To conduct training, five populations of 256 individuals were created where the first 8 genes were randomly generated and the second 8 genes were the best return chromosome. 512 generations of evolution were carried out. For each generation, each individual was evaluated with the same 10 randomly generated starting positions. Individual fitness values were based on the average fitness value in the 10 runs. The average was raised to the 1.5 power to amplify differences in fitness values between different individuals in a population. The individual with the best fitness was included in the next generation automatically. To fill the rest of the next generation, the area coverage section of the remaining individual chromosomes was produced through applying the three genetic operators: selection, crossover and mutation. Population details were saved during the evolution at several generations from 0 up to 512.

For a single run an individual was assigned a fitness value corresponding to the number of squares it reached. The simulated robot has 600 steps in which to cover the colony area, which was divided into 100, 50×50 unit squares, with the center 38×38 units being the contact width of each square. To reach a square, the robot's x and y coordinates have to reside in the contact area of a square. During an individual's simulation run each new square reached was recorded and the total number of the squares reached represented the robot's fitness value. If the simulated robot reached within 25 units of a wall the run would terminate. This made it difficult for the robot to reach the outer layer of squares. In training, any of such squares were counted if reached but for the final fitness evaluation tests those squares did not count. The resulting 64 colony area squares are shown in Fig. 13.

Run termination and fitness value assignment could occur for four reasons: all the squares in the area were reached, the robot took the maximum number of steps, the robot got closer

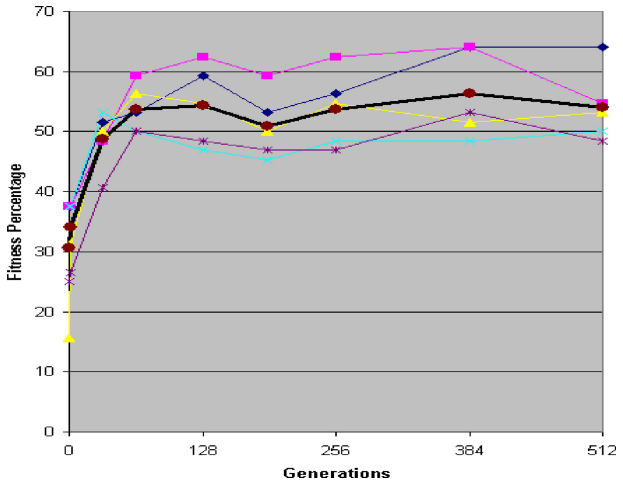

Fig. 14. Fitness of the complete self-sufficient behavior. For each population the best individuals at each generation were recorded (colored lines). The black bold line represents the average of the five tests.

than 25 units of a wall, or the robot's power level dropped below the empty power threshold of resulting in discontinued movement.

A. Results

For each of the five tests (one for each randomly generated population), analysis was done on the population details for 0, 1, 32, 64, 128, 256, 384, 512 generations. The populations at each of these generations were evaluated from 100 randomly generated start positions. Results of the best individual fitness (averaged over the 100 trials) from each population at each stored generation were recorded (Fig. 14). The results of all five runs of the complete task performance show that the population fitness average quickly grew through 64 generations, beyond which it modestly improved to a maximum at 512 generations.

Observations of the simulation revealed that in each population, the individual with the best fitness would behave in the following manner. After a charging routine the robot would rotate in one direction to face the colony area. It would then cover the area with a wide arc pattern through a continuous pattern of minimal turn gaits. If it got too close to a wall the turns angles increased to avoid getting much closer; after which, the robot continued to make minimal turns. At the time when robot reached an opposite wall or got far enough from the charging station, it would switch back to return mode.

Because the robot started a charging routine in a different angle/position each time, area coverage sweep variation was generated. In each area coverage sweep, the robot avoided the dark area when about to switch into return mode to ensure that it could use its light sensors to reach the charging station. The individual chromosome with the best fitness from the 512th generation for each of the five populations was tested from one specific colony area position ($x = 415$, $y = 252$, $h = 98$). The coordinates represent a possible robot position after a charging routine. The robot made wide overlapping sweeps of the colony area in each test, although there were significant differences in strategy. In tests 1 and 3, the robot stayed within the center of the area and made minimal left turns. In test 2, the robot stayed in the upper half of the colony area and made minimal left turns. In test 4, the robot makes sharp right turns. In test 5, the robot alternated between staying in the upper half of the area where it made minimal right turns, in the lower half of the area where it made minimal left turns.

This section has showed that using a multiloop CGA with conditional branching in incremental steps is an effective method for learning the complete self-sufficient tasks for our simulated environment of the hexapod colony robot.

VII. Tests on the Actual Robot

A key factor in this research was that the behavior learned in the simulation would be transferable to the physical robot. It was essential that the controller developed for the robot would be successful in the physical colony space with the actual power supply system. Tests on the robot resulted in modifications to the robot and adjustments to the simulation. The changes to the physical robot and changes to the simulation will not be discussed in this paper; the final version was presented in Section II. Final tests on the actual robot were conducted to validate our overall self-sufficient approach as specified by the hardware and simulation setup.

A. Transferring Simulated Behavior to the Physical Robot

The first step in developing a main controller program was to use a conversion program to convert a given chromosome into a series of instructions understandable by a BASIC Stamp II. The program was written in Scheme, and would convert the chromosome into a series of BASIC instructions. Instructions were sectioned off by their specific gene so as to create a method of branching to a specific gene's instructions. Instruction types included a move instruction and a branch instruction.

To facilitate the execution of a move instruction, the number type (i.e., 0–15) of the gait to be executed was stored, and control of the program branched to the 'move' section of the program. Once in the move section, the four bits making up the gait number to be executed were used to set the four output states of the pins of the main controller. The output pins of the main the controller were connected by wires to the four input pins on the walking controller so that the walking controller could read and execute the transmitted gait. The output states were held for 1600 ms (the approximate amount of time needed to execute a single gait cycle), after which control return back to gene so that the next instruction of the gene could be executed.

To facilitate the execution of a branch instruction, the sensor states test (i.e., 0–7) and the branching address (i.e., a specific gene) were stored, and control of the program branched to the "branch" section of the program. In this section, the specific sensor state to be tested was tested by reading the input of the specific sensors involved. If a test turned out to be positive, control of the program would jump to the specific gene specified in the branching address. Otherwise, control returned to original gene so that the next instruction could be executed.

The execution of instructions formed the main function of the program and the control of the program's instruction always resided in one of four general states. These four states represented the self-sufficient cycle and control could only change in the correct order of the cycle. Throughout the repeated cycles, a maximum of 600 gait cycle steps could be executed by the walking controller (representing the number of steps in a run).

For the beginning of a run, control would start in the sitting state, where the robot was charging at the power station. The main controller would send the "sit" gait output in its four output pin to the walking controller, and the main controller program would continue to do so until it received a signal from the ADC that charging was complete. Once the voltage sensor input indicated that charging was complete, control would switch from the sit state into the move backwards state. In this state, the four output pin states were set to the "straight backward" state, and were held for 4500 ms thus allowing the walking controller to read and execute 3–4 backward gait cycles. After the 4500 ms, control was sent into the area coverage mode, i.e., the first eight genes.

Control would start in the first gene, where the gene's four instructions were executed for the specified number of repetitions expressed in the gene structure. If there were not infinite repetitions or if there was not a successful branch to another gene, control was passed to the next gene once the instructions were completed. If the control needed to pass to the next gene but the current gene was the 8th gene, control would return to the first gene. This procedure continued until a low-power sensor test tested positive, in which case, control would shift to return mode, i.e. the second eight genes. The flow of instructions would start in the 8th gene and would flow through the second eight genes in the same manner as the flow in the first eight gene implemented in the area coverage mode. Control would last in the return state until a voltage jump occurred. Voltage jumps were tested for in each move and branch instruction. Once one such voltage jump was detected, control would return to the sit state and the four state cycle was completed.

B. Readjustments to the Simulated Learning

Once the several actual robot tests were tested in the physical colony space, it became evident that some readjustments to the simulated learning were needed. These issues were resolved and the newer solutions were already presented as part of the original discussion. However, they are important to mention because they help to identify significant factors not modeled in the simulation.

Firstly, there were issues with the fact that the IROD sensors and the metallic probes on the physical robot were situated in the front of the robot, while the simulation assumed that they were situated in the middle of the robot. This assumption existed in part because the simulated robot's xy coordinates formed its entire space in simulation and the length and width of the robot were not specified. An approximate solution involved increasing the length of the IROD sensors activation distance by 15.5 units (i.e., the distance from the IROD sensors to the center of the robot) and the length of the metallic probes in simulation by 8 units (the distance from the probes to the center to the front of the robot). In this way, the actual robot was often more able to sense walls and turn in time before crashing into a wall, and the metallic probe would be allowed greater flexibility in making contact with the power station.

Secondly, there was an issue with the light source and sensors in simulation that needed to be adjusted. The light was emitted from a single point and the light sensors range of vision did not overlap in simulation. As a result, both light sensors could not concurrently sense the light source. This was important problem to remedy because having both light sensors sense the light source was needed to perform effective return behavior. To remedy this, the simulation was modified so that two light source points were used and the light sensors were set to partially overlap.

C. Readjustments Required for the Actual Environment

The initial actual tests also made it clear that adjustments were needed in the actual environment. In the actual tests, as the robot would approach the charging station, there would be occasions where full contact would not be attained. This was typically due to the amount of angle between the probes and the plates of the charging station. When the probes were 90° to the surface of the plates, proper contact was assured. However, as this angle increased or decreased significantly, the probability of proper contact decreased.

Our solution for this issue will be addressed in the next phase of research when we set up circular charging stations that will have capacitors that are charged by solar panels. In this way, the charging stations can be autonomous and will not be placed along the walls since their circular nature will allow them to be placed anywhere in the colony space. The planned objective is to get them nearer to the workspace, but the other advantage is in relationship to the angle of probe contact. Because of the circular nature of the charging stations, the angle of the probe to the charging plates will almost always be close to 90°. With this in mind, during the experiments of this phase of the research, we allowed occasional hand guiding of the robot to facilitate full probe/plate contact when it was evident that proper contact was not being made.

D. Actual Tests

A series of actual robot tests were conducted to confirm the viability of the CGA for learning in our self-sufficient system. It was desired to test different behavior patterns learned over the area coverage evolution. The population whose best individual had the most noticeable improvement in behavior over evolution was selected. This best-individual behavior was evaluated at the 0, 32, 128 and 512 generations, such that one run was conducted from the starting coordinates of $x = 415$, $y = 252$, and $h = 98$. The performance (in simulation) of the specific population chosen is shown in Fig. 15. It is evident from the figure that the behavior in this population improved steadily; the best-individual robot gradually learned to make wide, partially overlapping sweeps of the area, while hitting walls less frequently and while

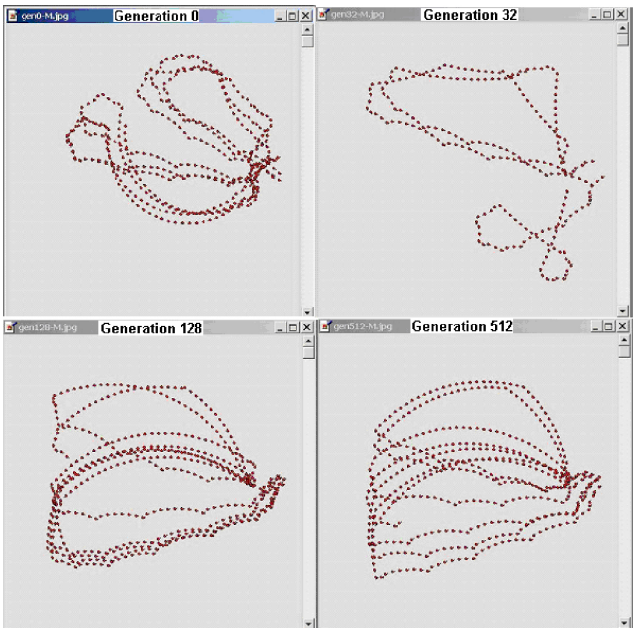

Fig. 15. Performance of the best individual throughout selected generations in the evolution. The population was selected due to its steady improvement in self-sufficient task performance.

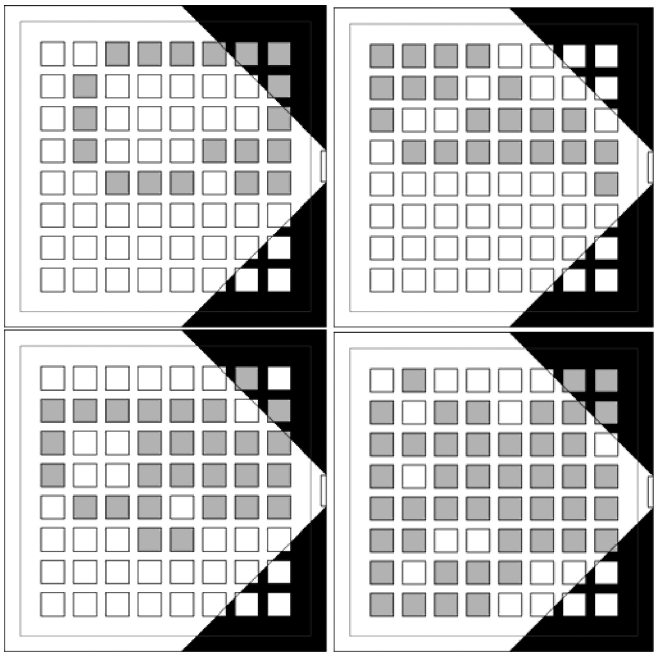

Fig. 16. Evaluation of the best individual chromosome transferred to the actual robot. The best from the generation 0 (top left), 32 (top right), 128 (bottom left), and 512 (bottom right) are shown.

making use of the variations in postcharging routine positions to get aligned to cover new area. This steady improvement of evolved behavior was reason enough for us to select the population and its best individual for testing on the actual robot.

In order to execute the tests, the best individual chromosome of the chosen population at each of the 4 generations (0, 32, 128, 512) was converted into a main controller program and tested three times on the actual robot. Evaluation in the physical colony space was facilitated by the visual observation—squares were drawn on the physical 8×8 ft colony floor (in the correct scale), and any squares reached during a test run were counted and recorded. Each run was started with the robot in charging mode, and the self-sufficient cycle was continuously repeated until the walking controller counted 500 gait cycles, or until there was a wall collision. Physical contact with the charging station was facilitated by hand to guide the robot when it was evident that the robot would not make proper contact. This happened for various reasons: premature energy loss, faulty robot headings, lack of precise charging station connection mechanisms, wrongly/undetected jump signals, etc.

VIII. Results of Actual Robot Tests

Three trial runs were recorded for each generation. The results showed an increase in fitness from one generation to the next with the average blocks covered at each generation: gen0: 11.3, gen32: 12.3, gen128: 20.7, and gen512: 34.0. The best from each of the generations is shown in Fig. 16 for comparison with the simulated results in Fig. 15. As can be seen, there are some similarities between the simulation and actual tests. In both cases, the track does not look bad at generation 0, but the robot's motions are too repetitive. It appears to be in an explore-for-solution phase at generation 32. By generation 128, it starts to develop a good pattern and this appears to be more closely perfected by generation 512.

While the actual robot tests do not necessarily coincide precisely with simulated robot tests, it is still noticeable that significantly more area was covered in each generation. One factor in producing such an improvement in the actual robot tests was the increase in amount of wide sweeps taken due to the increased ability of avoiding crashing into walls in later generations. This is similar to how the robot in simulation was not fully able to avoid walls when attempting to undergo charging routines in the early generations. It is also noteworthy, however, that the behavior in the later generations of the simulation did not feature as much wall following as in earlier generations. When the later generation behavior was transferred to the actual robot and tested, the likelihood of the actual robot crashing into a wall was reduced.

Intuitively, it would seem true that wall following with IROD sensors in the simulation was a relatively simple behavior that could be performed by the actual robot with success. However, discrepancies/malfunctions in actual robot commonly occurred and could be explained by a number of reasons: the loss of synchronous data transfer between microcontrollers, inadequate microcontroller battery levels, imperfect sensing capabilities/configurations, the gradual decrease in effectiveness of robot movement due to decreasing power storage, minor electrical circuitry issues, etc. While these issues affected and altered the planned behavior, they did so mostly in common occurrences on a small scale and were not highly significant in effecting overall behavioral patterns. The issues that could be addressed were corrected. When these operational issues were best minimized, the overall results of the actual robot tests indicated that the actual robot could

perform the self-sufficient task and compared relatively well to the behavioral performance in simulation. The success of the actual robot tests shows the viability of using a CGA to generate control programs for robots in a self-sufficient environment.

IX. Conclusion

Self-sufficiency is an important aspect of autonomous robots operating without consistent human contact. We have presented such a system that uses capacitors for power storage on a hexapod robot. Capacitors allow the robot to continually pursue the completion of its main task with recharge interruption times that are less than battery based systems. Although this method of self-sufficiency has its advantages, it presents a challenge for learning systems. Since both the charge and discharge times are reduced, the robot control program needs to check the power often and conduct its main task in such a way that it can always return to the charging station when low on power. An effective control program needs to balance the need to complete the task with the need to continually recharge. For the area coverage problem, it must move in such a way that it is in the proper position to return to the charging station and ensure that its trip back is of some benefit to the main task.

To accomplish this level of learning, we employed a CGA. Using the CGA for this application makes the learning system capable of learning area coverage control behaviors with less precise legged robots without adding hardware upgrades. In addition, since the CGA learns a program in the low-level control language of the robot, its results can be directly loaded to the control chip.

The CGA learning was done in two increments where in the first it learned a return program that would move the robot to the charging station and in the second it learned to complete the main task (area coverage) while using the return program as a subtask available to the area coverage program. Both increments of the CGA learning resulted in good control programs suitable for completing their task in simulation.

The control programs where then down loaded to the actual robot to verify the transferability of the programs learned in simulation. Some adjustments to the robot and the simulation were necessary after initial tests on the robot. Final tests on the actual robot confirmed the viability of the CGA centered learning system. Tests at four generations of learning showed similarities in effectiveness between the simulation and actual robot results. In addition, the learning system produced a control program that could successfully perform area coverage while maintaining self-sufficiency through timely sessions of recharging.

The approach to learning self-sufficient tasks through a CGA was successful and can be applied to other self-sufficient robot systems in which learning behavior is sought. The balance between when to perform ones primary task and when to stop working in order to recharge is difficult to determine. Learning systems such as the CGA can help in this regard. In addition, the use of incremental learning with the early increments being available as subtasks to the later increments can enhance evolution is a self-sufficient environment.

There are many possibilities for future work in this area. Several changes in the self-sufficient environment will provide opportunities to employ our incremental CGA learning approach to these new tasks. The capacitors are improving, with ones that are rated for more voltage and capacitance. New charging stations can be configured, requiring the robot to choose where to recharge. These charging stations could be using wind or solar power to gain their energy, requiring the robot to determine if it should go to a closer station with less power or to one further with more power, or possibly it should just wait. Additional robots can be added to the environment, requiring coordination between robots in order to effectively complete the task, plus keep enough power to run continually. In addition to these new challenges for a learning system due to improved or increased hardware, there are other possible future areas of research. One is to use the incremental CGA approach to learn new tasks, such as coordinated box pushing or predator/prey, within the self-sufficient environment. Another would be the use of alternate methods of evolutionary computation to learn control for the self-sufficient robot system.

References

[1] D. McFarland, "Autonomy and self-sufficiency in robots," in *The Artificial Life Route to Artificial Intelligence. Building Embodied, Situated Agents*, L. Steels, Ed. Mahwah, NJ: Lawrence Erlbaum, 1995, pp. 187–213.

[2] R. Brooks and A. Flynn, "A robot invasion of the solar system," *J. Brit. Interplanetary Soc.*, vol. 42, no. 10, pp. 478–485, 1989.

[3] D. McFarland and E. Spier, "Basic cycles, utility and opportunism in self-sufficient robots," *Robot. Autonomous Syst.*, vol. 20, nos. 2–4, pp. 179–190, 1997.

[4] J. Wawerla and R. Vaughan, "Optimal robot recharging strategies for time discounted labour," in *Proc. 11th Conf. Artificial Life*, Aug. 2008, pp. 670–677.

[5] G. Parker, R. Georgescu, and K. Northcutt, "Continuous power supply for a robot colony," in *Proc. WAC*, Jun. 2004, pp. 279–286.

[6] A. Birk, "Autonomous recharging of mobile robots," in *Proc. 30th Int. Symp. Automative Technol. Autom.*, 1997.

[7] G. Parker and R. Zbeda, "Controller use of a robot colony power supply," in *Proc. IEEE Int. Conf. SMC*, Oct. 2005, pp. 3491–3496.

[8] G. Parker and R. Zbeda, "Learning navigation for recharging a self-sufficient colony robot," in *Proc. IEEE Int. Conf. SMC*, Oct. 2007, pp. 734–740.

[9] G. Parker and R. Zbeda, "Learning area coverage for a self-sufficient colony robot," in *Proc. IEEE CEC*, May 2009, pp. 2083–2090.

[10] J. Walker, S. Garrett, and M. Wilson, "The balance between initial training and lifelong adaptation in evolving robot controllers," *IEEE Trans. Syst., Man, Cybern. B, Cybern.*, vol. 36, no. 2, pp. 23–432, Apr. 2006.

[11] X. Yao, "Evolving artificial neural networks," *Proc. IEEE*, vol. 87, no. 9, pp. 1423–1447, Sep. 1999.

[12] R. D. Beer and J. C. Gallagher, "Evolving dynamical neural networks for adaptive behavior," *Adaptive Behavior*, vol. 1, no. 1, pp. 91–122, 1992.

[13] D. Floreano and F. Mondada, "Evolution of homing navigation in a real mobile robot," *IEEE Trans. Syst., Man, Cybern. B, Cybern.*, vol. 26, no. 3, pp. 396–407, Jun. 1996.

[14] H. H. Lund and O. Miglino, "From simulated to real robots," in *Proc. IEEE 3rd Int. Conf. Evol. Comput.*, May 1996, pp. 362–365.

[15] Y. Zhou and M. Er, "An evolutionary approach toward dynamic self-generated fuzzy inference systems," *IEEE Trans. Syst., Man, Cybern. B, Cybern.*, vol. 38, no. 4, pp. 963–969, Aug. 2008.

[16] J. Busch, J. Ziegler, C. Aue, A. Ross, D. Sawitzki, and W. Banzhaf, "Automatic generation of control programs for walking robots using genetic programming," in *Proc. EuroGP 2002*, LNCS 2278. 2002, pp. 258–267.

[17] C. Lazarus and H. Hu, "Using genetic programming to evolve robot behaviors," in *Proc. 3rd Brit. Conf. Autonomous Mobile Robot. Autonomous Syst.*, 2001.
[18] P. Nordin, W. Banzhaf, and M. Brameier, "Evolution of a world model for a miniature robot using genetic programming," *Robot. Autonomous Syst.*, vol. 25, nos. 1–2, pp. 105–116, 1998.
[19] H. de Garis, "Genetic programming: GenNets, artificial nervous systems, artificial embryos," Ph.D. thesis, Univ. Libre Bruxelles, Bruxelles, Belgium, 1991.
[20] P. Petrovic, "Overview of incremental evolution approaches to evolutionary robotics," in *Proc. Norwegian Conf. Comput. Sci.*, 1999, pp. 151–162.
[21] E. Tuci, M. Quinn, and I. Harvey, "Evolving fixed-weight networks for learning robots," in *Proc. CEC*, 2002, pp. 970–1975.
[22] D. Bajaj and M. Ang, "An incremental approach in evolving robot behavior," in *Proc. 6th Int. Conf. Control, Autom., Robotics Vision*, 2000.
[23] A. Ram, R. Arkin, G. Boone, and M. Pearce, "Using genetic algorithms to learn reactive control parameters for autonomous robotic navigation," *Adaptive Behavior*, vol. 2, no. 3, pp. 277–305, 1994.
[24] G. Parker and R. Georgescu, "Using cyclic genetic algorithms to evolve multi-loop control programs," in *Proc. IEEE ICMA*, Jul. 2005, pp. 113–118.
[25] S. Yang and C. Luo, "A neural network approach to complete coverage path planning," *IEEE Trans. Syst., Man, Cybern. B, Cybern.*, vol. 34, no. 1, pp. 718–724, Feb. 2004.
[26] F. Zhang, Y. Xi, Z. Lin, and W. Chen, "Constrained motion model of mobile robots and its applications," *IEEE Trans. Syst., Man, Cybern., B, Cybern.*, vol. 39, no. 3, pp. 773–787, Jun. 2009.
[27] G. B. Parker, "Learning control cycles for area coverage with cyclic genetic algorithms," in *Proc. 2nd WSES Int. Conf. EC*, Feb. 2001, pp. 283–289.
[28] G. Parker, "Evolving cyclic control for a hexapod robot performing area coverage," in *Proc. IEEE Int. Symp. CIRA*, Aug. 2001, pp. 561–566.
[29] H. Choset and P. Pignon, "Coverage path planning: The boustrophedon cellular decomposition," in *Proc. Int. Conf. Field Service Robot.*, 1997.
[30] M. Reinstein and M. Hoffmann, "Dead reckoning in a dynamic quadruped robot: Inertial navigation system aided by a legged odometer," in *Proc. IEEE Int. Conf. Robot. Autom.*, May 2011, pp. 617–624.
[31] G. Parker, D. Braun, and I. Cyliax, "Evolving hexapod gaits using a cyclic genetic algorithm," in *Proc. IASTED Int. Conf. ASC,* Jul. 1997, pp. 141–144.
[32] *PIC12F629/675 Data Sheet*, Microchip Technology, Inc., Chandler, AZ, 2003.
[33] J. H. Holland, *Adaptation in Natural and Artificial Systems*. Ann Arbor, MI: Univ. Michigan Press, 1975.
[34] G. B. Parker and G. J. E. Rawlins, "Cyclic genetic algorithms for the locomotion of hexapod robots," in *Proc. World Autom. Congr. Robotic Manufact. Syst.*, vol. 3. 1996, pp. 617–622.
[35] G. B. Parker, I. I. Parashkevov, H. J. Blumenthal, and T. W. Guildman, "Cyclic genetic algorithms for evolving multi-loop control programs," in *Proc. WAC*, Jun. 2004, pp. 47–352.

Gary Parker (M'91–SM'12) received the B.A. degree in zoology from the University of Washington, Seattle, the M.S. degree in computer science from the Naval Postgraduate School, Monterey, CA, and the Ph.D. degree in computer science and cognitive science from Indiana University, Bloomington.

He is currently a Professor of computer science with the Connecticut College, New London, CT, where he is the Chair of the Department of Computer Science. His current research interests include methodologies for learning in autonomous agents. Currently, he is concentrating on evolutionary robotics and learning control programs for agents in real-time videogames.

Richard Zbeda received the B.A. degree in computer science and economics from the Connecticut College, New London, and the M.S.E. degree in computer and information science from the University of Pennsylvania, Philadelphia.

He is currently an Associate Engineer with Premier Technology Solutions, New York, working on software consulting projects. He has conducted research on artificial intelligence and software or database systems. His current projects with Premier Technology Solutions include enhancing Enterprise Search software for law firms.