Implementation of a Modular Framework to Compare Foraging Algorithm Effectiveness

Kelsey Geiger
School of Computer Science
California State University Channel Islands
Email: kelseyrgeiger@gmail.com

***Keywords*—ARGoS, iAntz, ROS, Gazebo, rovers, forgaging algorithm, swarm robotics**

I. Introduction

Swarm robotics is a relatively new and dynamic field, looking to learn how to control massive numbers of robots to complete tasks that cannot be solved by a single robot alone. Foraging problems are problems in which a set of rovers searches for resources of some sort and returns them to a collection location. Many foraging algorithms are inspired by social insects, especially ants. Hecker and Moses have described a Central Place Foraging Algorithm (CPFA) based on the foraging behavior of several ant species.[1] Another foraging algorithm developed by Fricke et al, called the Distributed Deterministic Spiral Algorithm (DDSA), is designed to cover the entire search area and collect targets in optimal time.[2]

These two algorithms had previously been implemented in ARGoS and on iAntz robots.[1][2] Moses and Fricke wished to compare the performance of these two algorithms under similar conditions. However, their implementations were too different to be able to isolate the algorithms themselves from changes to e.g. the driving, pickup, and dropoff routines for collecting resources. They also wanted to test the algorithms in a more realistic environment that better modeled the actual conditions rovers would operate under.

To these ends, the algorithms were written in a new framework, on the Robot Operating System (ROS) platform, simulated in Gazebo. The Gazebo simulator offers a much more realistic simulation of the physics of rover motion, including more accurate physics calculations and more detailed interactions. The ROS platform can be used for both simulated and real robots, and so it allows for code to be written once, tested in simulation, and then run on physical robots. The new framework, which is based off of code used in the NASA Swarmathon competition, shall be described in this paper.

II. Background

Previously, the code written to implement the CPFA and the code written to implement the DDSA were separately developed, so similar functions were fulfilled with different code. These differences were **not** insignificant- they led to increased efficiency in one algorithm over the other only because there were fewer collisions on the return path, due to different motor controlling code for each algorithm, and not inherent properties of the algorithms themselves. Because of this, there was an incentve to develop a common framework for the common tasks carried out by these real and simulated robots, in order to compare the algorithms directly.

The NASA Swarmathon competition, overseen by the Biological Computation Lab of University of New Mexico (BCLab-UNM), also provided the incentive to create a basic framework that would allow others to build their own algorithms off a set of basic utilities. The base version of this code, however, had its own algorithm, a simple random walk, and features of

that algorithm persisted through many different components of the monolithic "mobility module". This design was useful for rapidly producing a basic set of code that could be modified in a competition setting, but it was not good for software development purposes- it was not modular or easy to modify key components of this base code.

My primary task over the course of my DREU internship was to take this base code and convert it to a more modular and easily editable form, separating out key components and redefining the structure of the program to allow modification of only the algorithm used. Additionally, this code could be used for future NASA Swarmathon competitions. My secondary task during my DREU internship, once the primary had been completed, was to use this new framework to implement the DDSA and test it both in simulation and in physical rovers.

III. The New Design

The new modular framework designed over the course of the internship was inspired by interrupt handlers in operating systems. Different modules could interrupt the current action being done by the rover based on input. The modules each had a configurable priority, defined by a signed integer value, for their interrupts, based on the overall state of the rover's algorithm. In one state, some modules might be disabled, and in another they could take the highest priority. If multiple modules had interrupts that needed to be handled, then the highest-priority module would be attended to.

To make this framework compatible with C++, we defined a Controller interface, which each module implemented. We defined a PrioritizedController struct that was a pointer to a Controller and an integer priority value. These were stored in a standard priority queue, ordered by decrasing value (so that the highest priority value controller would be checekd first). There was a LogicController class used to manage the state of the overall program, including the management of the interrupt handling system. Every tenth of a second, all the Controllers besides this LogicController were polled in order of their priority. If any of them had "work" to do, they were given control of the rover until they no longer needed to issue motor commands ("do work").

These controllers were separated into several disjoint functions, each independent of the others. They were the SearchController, which defined which point to move to if no other action had to be taken; the DriveController, which actually carried out the movement of the rover, either through a waypoint system or a direct motor command; the PickUpController, which handled the picking up of cubes when they were sighted; the DropOffController, which handled returning to the collection zone once a cube had been picked up; and finally the ObstacleController, which handled collision avoidance and avoidance of cubes if one was already held.

A. SearchController

SearchController was the lowest priority Controller in the collection, and was effectively only allowed to issue motor commands if the previous end goal point was reached and no cubes were found along te way, or if a cube was found and dropped off and a new search point was needed. This controller effectively managed the search pattern for the rover, and as such was most key to testing different search algorithms.

B. DriveController

DriveController was not directly used in the priority queue system. It handled the actual translation of commands from other controllers into proper motor signals, managing PID controllers for the wheels and gripper and sending the appropriate messages out to a translation module (in this case, with ROS, but it could be used with other frameworks as well). It took a set of waypoints, or direct velocity values for either the wheels or gripper from the other controllers, and would translate them

into values to send to the motors and servos as appropriate.

C. PickUpController

PickUpController handled the tracking and picking up of cubes on the field. It took camera and ultrasound sensor input to locate blocks, drive to them, pick them up with grippers, and determine if the block was successfully held.

D. DropOffController

DropOffController handled the return to the collection zone and the process of dropping off the held cube. It took camera input to find the tags lining the collection zone. Once the collection zone was found, DropOffController aligned the rover towards the center, drove past the tags, and opened the gripper to release the cube, then backed up the rover. This would put it just outside the collection zone so that the state machine could cycle back to the first state.

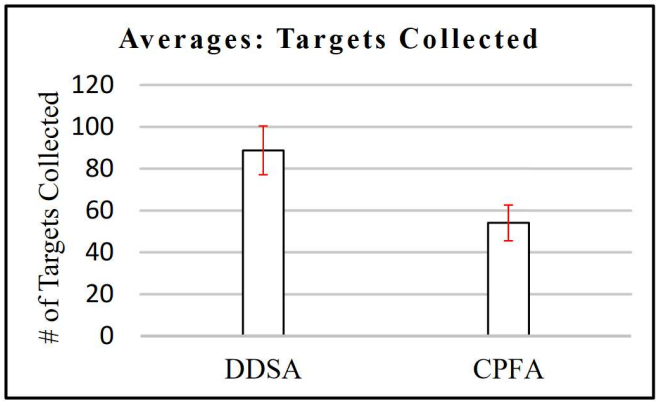
E. ObstacleController

ObstacleController handled the avoidance of various types of obstacles, depending on the current state of the program. During the search portion, ObstacleController took camera input and ultrasound sensor input to avoid the collection zone tags and other rovers and walls, respectively. During the return and dropoff portions, ObstacleController took camera and ultrasound sensor input to avoid cube clusters and other rovers and walls, respectively.

This framework allowed for much greater flexibility, since different modules could be kept the same and only SearchController could be modified to truly compare different algorithms side-by-side.

IV. Preliminary Testing

While more thorough testing could not be completed by the end of the internship, some preliminary testing was done. Both the CPFA and DDSA were implemented with the new system, and both were tested for 15 trials of 6 rovers running for 20 simulated minutes per trial. The DDSA averaged 88.73 cubes over the 15 trials while the CPFA averaged 54.07 cubes over 15 trials.



For the number of rovers used, this relative difference matches that found by Fricke et al.[2] However, given the limited number of trials done during this preliminary testing, these results are not conclusive, and future work must be done to achieve more accurate results. Additionally, the simulation should be able to scale to well over six rovers at a maximum, to arbitrary values of rovers, to better determine the limits of each algorithm and better understand how each scales with swarm size.

References

[1] Hecker, Joshua P., and Melanie E. Moses. "Beyond pheromones: evolving error-tolerant, flexible, and scalable ant-inspired robot swarms." *Swarm Intelligence* 9.1 (2015): 43-70.

[2] Fricke, G. Matthew, et al. "A distributed deterministic spiral search algorithm for swarms." *Intelligent Robots and Systems (IROS), 2016 IEEE/RSJ International Conference on*. IEEE, 2016.