Evolved Foraging Strategies

Nicolas Ward
nward@fas.harvard.edu

Andrew Reiter
areiter@seas.harvard.edu

Pierre-Emile Duhamel
pierre.duhamel@gmail.com

Harvard University
CS266 - Prof. Rhadika Nagpal

2012.05.03

Abstract

Our CS266 Final Project applies various techniques to succeed at the task of competitive multi-robot foraging. We used three manually programmed e-puck robots, with custom camera image-processing code and local sensing to complete the project milestones. In Milestone 1, foraging, our team collected 13/15 food items, tieing for first place. In Milestone 2, competitive foraging, our team scored 8-8 and 3-6. The decrease in performance was due primarily to increasing sensitivity of the range sensors, although no changes to the code were made between trials. For the simulation task, we used genetic programming (GP) to evolve novel multi-robot strategies. We determined reasonable ranges for the mutation rate and population size GP parameters, and evolved numerous strategies that were fitter than the baseline. We additionally experimented with using an Extended Kalman Filter for global localization on the robots, and with converting evolved step programs to C code that could run on-robot. These experiments were not complete in time for Milestone 2, but would make an interesting area of future work.

1 Introduction

We pursued a two-pronged approach to solving the problem of competitive multi-robot foraging defined by the CS266 tournament competition using e-puck robots. We developed an image segmentation algorithm that worked well within the robot's limited memory, and calibrated the range sensors for each individual robot. Our simulation implementation attempted to replicate this configuration virtually, allowing us to evolve strategies. We also tested an Extended Kalman Filter to perform state estimation.

1.1 Related Work

In order to test and develop strategies for the robots, we use the MASON Simulation Toolkit [6] for Java to represent the foraging tournament field used in Milestones 1 and 2. Development progress was driven by the MASON library's manual [5] and included tutorials.

Initial manual strategies were developed based on experimental results from preparing for the project milestones, as well as past experience with robots performing urban search & rescue (USR) tasks [8]. These manually written strategies also served as the seeds of our strategy optimization.

Search and optimization algorithms have long been a staple of artificial intelligence research. Adaptive search methods, including evolutionary approaches such as genetic algorithms, have been in use since the late 1970s [3]. Genetic programming [4] (GP) is a relatively recent innovation that is well-suited to improving strategies for multi-robot tasks, such as the annual RoboSoccer competition [7].

We designed our "step program" format with GP in mind, taking inspiration from functional paradigm languages such as LISP to define our operators as S-expressions [9]. The overall design of our crossover and mutation techniques was based on Chapter 3 of Fundamentals of Natural Computing [2], which we previously read for class.

2 Robot Implementation

The e-puck robot [11] offers many hardware options including a color camera, ring of proximity sensors, a range and bearing communication turret, speaker, three microphones, accelerometer, and two stepper motor controlled wheels in a compact robotic device. A 3D printed kicker was added to the e-puck to allow it to move red pucks representing food across the field to a yellow or green nest. In the following subsections, we describe the algorithms implemented on the e-puck robot for the milestone 1 and 2 foraging competitions.

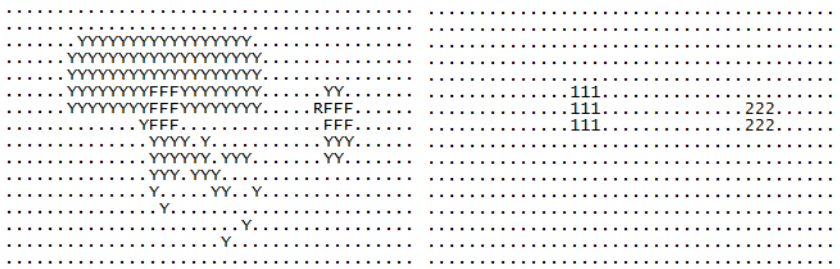
2.1 Vision Algorithms

The e-puck camera is a forward looking, 640x480 resolution, color camera. Each frame is down sampled to 80x15 resolution and cropped to the lower 10 rows. This restricts the processing of vision algorithms to pixels sensing the ground up to the walls of the arena. The various food and goal objects are detected visually. A threshold can be applied to pixels in the image to classify the pixel as belonging to an object of interest or not (Table 1).

Color thresholding (Figure 1(a)) is able to detect pixels in an image that appear to represent food, but it does not differentiate between images with a single, contiguous appearance of food and images with multiple, discontinuous appearances. We use component image segmentation to create separate masks for each distinct food that appears in an image (Figure 1(b)). The algorithm

Object	Red	Green	Blue
Food	> 10	< 10	
Green Goal	< 4	> 8	< 1
Yellow Goal	> 10	> 15	

Table 1: Thresholds for Objects



(a) Thresholded image (b) Component image

Figure 1: An example thresholded image in shown in (a), with "Y" representing pixels corresponding to the yellow goal and "F" representing pixels corresponding to food. The goal pixels that appear below the ninth row from the bottom are noise. Goal noise is avoided by only searching the top four lines for the goal. The corresponding food component image is shown in (b), clearly segmenting the two foods into component 1 and component 2, so that a distance and angle can be calculated to each.

used to generate the component image is given in Algorithm 1 and uses a two-pass scheme to join non-distinct segments that were marked as distinct objects. The algorithm is only valid for round or rectangular shapes; a third or fourth pass would be necessary to test the connectedness of arbitrary shapes.

To detect the distance and angle to an object, whether food or goal, the left-most and right-most edges of the object are detected. For the goal, the original image is analyzed, whereas for food, the mask corresponding to a specific component is analyzed. Note that even if the middle of a goal is obscured, the width and midpoint can still be recovered correctly, because the left-most and right-most edges are still visible. We also only process the top 2 rows of the image to find the goal, which is more efficient and looks overhead any robots at the goal. The distance d is then computed as

$$d = \frac{f}{L_e - R_e} W$$

where f is the camera focal length as a multiple of the width of a pixel, L_e and R_e are the numbers of columns from the left of an image in which the left and right edges of an object were detected, respectively, and W is the physical width of the object projected along the optical axis. We found the e-puck focal length to be $619.00/X_{zoom}$ pixel widths, where X_{zoom} is the camera horizontal

subsampling factor. The angle θ to the object is similarly computed as

$$\theta = \tan^{-1}\left(-\frac{1}{f}\left(\frac{L_e + R_e}{2} - x_{PP}\right)\right)$$

where x_{PP} is the number of columns from the left of the image of the principal point.

Algorithm 1 Generation of component image which segments and masks non-overlapping food occurrences

```
for r = 2 → height do
    for c = 2 → width do
        if image(r, c) is food then
            if image(r − 1, c) is food then                ▷ part of existing object
                component(r, c) ← component(r − 1, c)
            else if image(r, c − 1) is food then           ▷ part of existing object
                component(r, c) ← component(r, c − 1)
            else                                           ▷ possible new object
                maxObjects ← maxObjects + 1
                component(r, c) ← maxObjects
            end if
        else
            component(r, c) ← BACKGROUND
        end if
    end for
end for
for r = 2 → height do                  ▷ second pass joins non-distinct neighbors
    for c = width − 1 → 1 do
        if image(r, c) is food AND
          image(r, c + 1) is food AND
          component(r, c) ≠ component(r, c + 1) then
            component(r, c) ← component(r, c + 1)
        end if
    end for
end for
```

2.2 Foraging Strategy

All three members of our robot collective run the same code. By having all robots do the same tasks, we take advantage of the parallelism of the collective. The algorithm is structured as a finite state machine and described in Algorithm 2.

In the find food state, the robot rotates as fast as possible in 45° steps, such that the new view has a slight overlap with the previous view. Since we can generally see food from across the field, the robot makes a straight line movement

Algorithm 2 Finite state machine running on all robots.

```
loop
    switch currentState do
        case findFood
            if foodInView then
                MOVE(stop)
                currentState ← gotoFood
            else if circleComplete then
                MOVE(line)
            else
                MOVE(steppedCircle)
            end if
        case gotoFood
            if !foodInView then
                currentState ← findFood
            else if foodInGrip then
                currentState ← findGoal
            else
                MOVE(dist2Food, angle2Food)
            end if
        case findGoal
            if !foodInView then
                currentState ← findFood
            else if goalInView then
                MOVE(stop)
                currentState ← gotoGoal
            else if circleComplete then
                MOVE(line)
            else
                MOVE(smoothCircle)
            end if
        case gotoGoal
            if !foodInView then
                currentState ← findFood
            else if !goalInView then
                currentState ← findGoal
            else
                MOVE(dist2Goal, angle2Goal)
            end if
end loop
```

away from its current position if it does not see food within approximately one rotation to search from a different viewpoint on the field. Once food is found, the robot travels to the food. The distance to the food, found as discussed

previously, is used as a threshold to determine if food is in the gripper. Next, the robot finds the goal similarly to finding the food, but rotates in a smooth circle, rather than the stepped circle used to find food, so that food is not lost from the gripper. Once the goal is found, the robot continuously moves towards it.

2.3 Collisions

At first we tested our foraging strategy without collision avoidance and found that, almost immediately, at least two of the robots would collide such that they could not separate. The most severe modes of failure were collisions where the gripper of one robot gets stuck in the wheel or gripper of another robot. Originally we assumed that robots could not be broken up in competition, such that disabled robots would give a significant loss of performance and a strategy to avoid collisions was necessary.

Our strategy used the proximity sensors, rather than the range and bearing communication ring, so that we can also recover from getting trapped in a corner. The collision avoidance algorithm is implemented within the Move function call in Algorithm 2. Once the robot needs to execute a move, all the proximity sensors are checked and the single proximity sensor reporting the closest object is checked against its threshold. If the threshold is violated, the robot will execute a collision avoidance move consisting by driving with a set linear and angular speed for a time period. The linear speed is fixed at 800 units of e-puck speed, but the forward backward direction is opposite the front/back proximity sensors or randomly chosen for side sensors. The angular speed is randomly chosen between -200 and 200 units of e-puck speed and the time period is randomly chosen between 100ms and 600ms. This results in the robot moving randomly away from its obstacle, which allows it to re-approach the situation from a different state.

2.4 Communication

We believed that communication would become difficult in a competitive game with 6 robots on the field. Unless the ranges of data communication are restricted for each team, it would be easy for the opposing team to interfere with a communication based strategy. One possible method of attack would be simply echoing back what each robot hears. We decided it would be interesting to attempt to get great performance without relying on communication and did not use communication for any milestones.

2.5 State estimation

Many strategies benefit by using the robot's pose (position and orientation) and an input rather than current sensor readings. To abstract pose from sensor readings, we developed an extended Kalman filter (EKF) to synthesize all measurements into a single pose estimate. We define a state

$$\mathbf{x} = \begin{bmatrix} x \\ y \\ \theta \end{bmatrix}$$

and its covariance matrix

$$P = \begin{bmatrix} \sigma_{xx} & \sigma_{xy} & \sigma_{x\theta} \\ \sigma_{yx} & \sigma_{yy} & \sigma_{y\theta} \\ \sigma_{\theta x} & \sigma_{\theta y} & \sigma_{\theta\theta} \end{bmatrix}.$$

2.5.1 Encoder propagation

When the e-puck is driving, the state and covariance can be propagated forward in time using the wheel encoders. Defining the robot dynamics as $\dot{\mathbf{x}} = f(\mathbf{x}, \mathbf{u})$ and linearizing them as $\mathbf{F}_k = \mathbf{I} + \frac{\partial}{\partial \mathbf{x}} f(\mathbf{x}, \mathbf{u})\big|_{\mathbf{x}=\hat{\mathbf{x}}_k}$, the propagation is performed by the EKF equations:

$$\hat{\mathbf{x}}_{k+1} = \hat{\mathbf{x}}_k + f(\hat{\mathbf{x}}_k,\ \mathbf{u}_k) \triangle t$$
$$\mathbf{P}_{k+1} = \mathbf{F}_{k+1}\mathbf{P}_k\mathbf{F}_{k+1}^T + \mathbf{Q}_k$$

Since our motor controller sets the wheel velocity, we can model driving as one of four states: no motion, straight line translation, pure rotation, or following a circular arc (Figure 2).

In the case of no motion, neither the state nor the covariance is propagated at all. In the other cases, a nonlinear propagation [1] is used that integrates the infinitesimal linear propagations to form and update which does not depend on step size - the new state and covariance remain optimal for arbitrarily large step sizes. This allows the e-puck EKF accuracy to be high even when the estimation loop runs slowly. We first define R and L and the distances traveled by the right and left wheels, B as the wheelbase diameter, and σ_e as the per-distance propagation noise. We can thus define $\hat{\mathbf{x}}_{k+1}$, $\mathbf{Q}_k$, and $\mathbf{P}_k$ for each case:
Straight line translation of distance $D = R = L$:

$$\begin{bmatrix} x_{k+1} \\ y_{k+1} \\ \theta_{k+1} \end{bmatrix} = \begin{bmatrix} x_k \\ y_k \\ \theta_k \end{bmatrix} + \begin{bmatrix} D\cos(\theta_k) \\ D\sin(\theta_k) \\ 0 \end{bmatrix}$$

$$\mathbf{F} = \begin{bmatrix} 1 & 0 & -D\sin(\theta) \\ 0 & 1 & D\cos(\theta) \\ 0 & 0 & 1 \end{bmatrix}$$

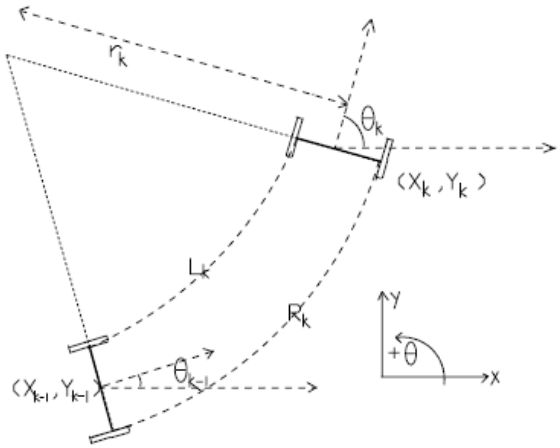


Figure 2: Initial and final state of the robot after following a circular arc trajectory [1]

$$
\begin{aligned}
Q_{1,1} &= |D| \left(\frac{1}{2}\cos^2\theta_k \ + \frac{2D^2}{3B^2}\sin^2\theta_k \right) \sigma_e^2 \\
Q_{2,2} &= |D| \left(\frac{1}{2}\sin^2\theta_k \ + \frac{2D^2}{3B^2}\cos^2\theta_k \right) \sigma_e^2 \\
Q_{3,3} &= \frac{2\,|D|}{B^2}\sigma_e^2 \\
Q_{1,2} = Q_{2,1} &= |D| \left(\frac{1}{4} - \frac{D^2}{3B^2} \right) \sin\left(2\theta_k\right) \ \sigma_e^2 \\
Q_{1,3} = Q_{3,1} &= |D| \left(-\frac{D}{B^2} \right) \sin\left(\theta_k\right) \ \sigma_e^2 \\
Q_{2,3} = Q_{3,2} &= |D| \left(\frac{D}{B^2} \right) \cos\left(\theta_k\right) \ \sigma_e^2
\end{aligned}
$$

Pure rotation of angle $\Delta\theta = \frac{R-L}{B}$:

$$
\begin{bmatrix} x_{k+1} \\ y_{k+1} \\ \theta_{k+1} \end{bmatrix} = \begin{bmatrix} x_k \\ y_k \\ \theta_k \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \\ \Delta\theta \end{bmatrix}
$$

$$
\mathbf{F} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}
$$

$$
\begin{aligned}
Q_{1,1} &= \frac{B}{16}\left[2\Delta\theta - \sin\left(2\theta_k\right) \ + \sin\left(2\left(\theta_k+\Delta\theta\right)\right)\right]\sigma_e^2 \mathrm{sgn}\left(\Delta\theta\right) \\
Q_{2,2} &= \frac{B}{16}\left[2\Delta\theta + \sin\left(2\theta_k\right) \ - \sin\left(2\left(\theta_k+\Delta\theta\right)\right)\right]\sigma_e^2 \mathrm{sgn}\left(\Delta\theta\right) \\
Q_{3,3} &= \frac{|\Delta\theta|}{B}\sigma_e^2 \\
Q_{1,2} &= Q_{2,1} = \frac{B}{16}\left(\cos\left(2\theta_k\right) \ - \cos\left(2\left(\theta_k+\Delta\theta\right)\right)\right)\sigma_e^2 \\
Q_{1,3} &= Q_{3,1} = 0 \\
Q_{2,3} &= Q_{3,2} = 0
\end{aligned}
$$

Circular arc of radius $R_c = \frac{B}{2}\frac{L+R}{L-R}$:

$$
\begin{bmatrix} x_{k+1} \\ y_{k+1} \\ \theta_{k+1} \end{bmatrix} = \begin{bmatrix} x_k \\ y_k \\ \theta_k \end{bmatrix} + \begin{bmatrix} R_c\left(\sin\left(\theta_k\right) \ - \sin\left(\theta_k+\Delta\theta\right)\right) \\ R_c\left(\cos\left(\theta_k+\Delta\theta\right) \ - \cos\left(\theta_k\right)\right) \\ \Delta\theta \end{bmatrix}
$$

$$
\mathbf{F} = \begin{bmatrix} 1 & 0 & -R_c\left(\sin\left(\theta_k\right) \ - \sin\left(\theta_k+\Delta\theta\right)\right) \\ 0 & 1 & R_c\left(\cos\left(\theta_k+\Delta\theta\right) \ - \cos\left(\theta_k\right)\right) \\ 0 & 0 & 1 \end{bmatrix}
$$

$$
\begin{aligned}
Q_{1,1} = \Bigg(& \frac{-4R_c}{(L-R)^2}(|R|L + R|L|)(\sin(\theta_k) - \sin(\theta_k + \Delta\theta))\cos(\theta_k) \\
& + 2\left(\frac{R_c}{B}\cos(\theta_k)\right)^2 (|R| + |L|) \\
& - \frac{B}{2(L-R)^3}[2(\Delta\theta - \sin(2\theta_k) + \sin(2(\theta_k + \Delta\theta)](L^2|R| + R^2|L|)\Bigg)\sigma_e^2 \\
Q_{2,2} = \Bigg(& \frac{-4R_c}{(L-R)^2}(|R|L + R|L|)(\cos(\theta_k + \Delta\theta) - \cos(\theta_k))\sin(\theta_k) \\
& + 2\left(\frac{R_c}{B}\cos(\theta_k)\right)^2 (|R| + |L|) \\
& - \frac{B}{2(L-R)^3}[2(\Delta\theta + \sin(2\theta_k) - \sin(2(\theta_k + \Delta\theta)](L^2|R| + R^2|L|)\Bigg)\sigma_e^2 \\
Q_{3,3} = & \frac{2}{B^2}(|R| + |L|)\sigma_e^2 \\
Q_{1,2} = Q_{2,1} = \Bigg(& -\frac{2R_c}{(L-R)^2}(|R|L + R|L|)(\cos(2(\theta_k + \Delta\theta)) - \cos(2\theta_k + \Delta\theta)) \\
& + \left(\frac{R_c}{D}\right)^2 \sin(2(\theta_k + \Delta\theta))(|R| + |L|) \\
& - \frac{B}{2(L-R)^3}[\cos(2\theta_k) - \cos(2(\theta_k + \Delta\theta))](L^2|R| + R^2|L|)\Bigg)\sigma_e^2 \\
Q_{1,3} = Q_{3,1} = \Bigg(& \frac{2}{(L-R)^2}(|R|L + R|L|)(\sin(\theta_k) - \sin(\theta_k + \Delta\theta)) \\
& - \frac{2R_c}{B^2}\cos(\theta_k + \Delta\theta)(|R| + |L|)\Bigg)\sigma_e^2 \\
Q_{2,3} = Q_{3,2} = \Bigg(& \frac{2}{(L-R)^2}(|R|L + R|L|)(\cos(\theta_k + \Delta\theta) - \cos(\theta_k)) \\
& - \frac{2R_c}{B^2}\sin(\theta_k + \Delta\theta)(|R| + |L|)\Bigg)\sigma_e^2
\end{aligned}
$$

The encoder noise σ_e^2 represents the standard deviation of distance traveled as a fraction of the square root of distance traveled. Over seven trials driving 600 mm, the e-puck stopped within 1 mm of the mean each time, giving a conservative estimate of $\sigma_e^2 = (1/\sqrt{600})^2 = 1/600$.

2.5.2 Sensor updates

The process noise associated with propagating the state forward increases the uncertainty, but sensor updates decrease it. Given a measurement model $\mathbf{z} = h(\mathbf{x})$, which describes the measurement $\mathbf{z}$ that a sensor would ideally return when the robot is in state $\mathbf{x}$, its linearized form $\mathbf{H}_k = \frac{\partial}{\partial \mathbf{x}} h(\mathbf{x})\big|_{\mathbf{x}=\hat{\mathbf{x}}_k}$, and the

expected measurement variance, $\mathbf{R}$, the EKF equations update both the state and its covariance:

$$\begin{aligned}
\hat{\mathbf{z}}_k &= h\left(\hat{\mathbf{x}}_{k^-}\right) \\
\widetilde{\mathbf{z}}_k &= \mathbf{z}_k - \hat{\mathbf{z}}_k \\
\mathbf{S}_k &= \mathbf{H}_k \mathbf{P}_{k^-} \mathbf{H}_k^T + \mathbf{R}_k \\
\mathbf{K}_k &= \mathbf{P}_{k^-} \mathbf{H}_k^T \mathbf{S}_k^{-1} \\
\hat{\mathbf{x}}_{k^+} &= \hat{\mathbf{x}}_{k^-} + \mathbf{K}_k \widetilde{\mathbf{z}}_k \\
\mathbf{P}_{k^+} &= \left(\mathbf{I} - \mathbf{K}_k \mathbf{H}_k\right) \mathbf{P}_{k^-}
\end{aligned}$$

where $\mathbf{z}_k$ is the measurement returned by the sensor, $\widetilde{\mathbf{z}}$ is the measurement residual, $\mathbf{S}$ is the measurement prediction covariance, and $\mathbf{K}$ is the Kalman gain.

The camera is used as a sensor to measure the angle to known reference points in the field. For these reference points, we used "goalposts," the blue-to-yellow, yellow-to-blue, blue-to-green, and green-to-blue edges of the goals. Each has a distinct visual edge and a known absolute location. The measurement z is the number of pixels from the left of the image that a goalpost appears. Using the camera projection equations

$$\frac{z - x_{PP}}{f} = \frac{X}{Z},$$

where x_{PP} is the number of pixels from the left of the image of the principal point, f is the focal length as a multiple of pixel widths, Z is the altitude along the optical axis to the plane containing the goalpost, and X is the distance between the goalpost and the altitude's foot, it is possible to define the expected measurement z as a function of the state $\hat{\mathbf{x}}_{k^-}$ and goalpost coordinates (x_{GP}, y_{GP}). Defining the camera's location as

$$\begin{bmatrix} x_c \\ y_c \\ \theta_c \end{bmatrix} = \begin{bmatrix} x_k + r_c \cos\left(\theta_k\right) \\ y_k + r_c \sin\left(\theta_k\right) \\ \theta_k \end{bmatrix}$$

where r_c is the distance from the camera's center of projection to robot's center axis, which we estimate to be 3.2 cm, the resulting measurement model is

$$\hat{z}_k = x_p - f \tan\left(\tan^{-1}\left(\frac{y_{GP} - y_c}{x_{GP} - x_c}\right) - \theta_k\right)$$

$$H_1 = -\frac{(y_{GP} - y_c) f \left[\tan^2\left(\tan^{-1}\left(\frac{y_{GP}-y_c}{x_{GP}-x_c}\right) - \theta_c\right) + 1\right]}{(x_{GP} - x_c)^2 \left[\left(\frac{y_{GP}-y_c}{x_{GP}-x_c}\right)^2 + 1\right]}$$

$$H_2 = \frac{f \left[\tan^2\left(\tan^{-1}\left(\frac{y_{GP}-y_c}{x_{GP}-x_c}\right) - \theta_c\right) + 1\right]}{(x_{GP} - x_c) \left[\left(\frac{y_{GP}-y_c}{x_{GP}-x_c}\right)^2 + 1\right]}$$

$$H_3 = f \left[\tan^2\left(\tan^{-1}\left(\frac{y_{GP} - y_c}{x_{GP} - x_c}\right) - \theta_c\right) + 1\right]$$

The camera measurement variance comes from the image discretization due to the large pixel size. The true (continuous) pixel location of a reference point is, on average, distributed uniformly across the width of a pixel. Assuming no gaps between pixels, the width of a pixel is the inverse of the focal length in pixels, or $1/f$. Thus, the variance of this uniform distribution across a pixel is $R = \frac{1}{12}\left(\frac{1}{f}\right)^2$.

The proximity sensors measure the distance from the robot and the surrounding walls to update the robot's position and orientation. First, we determine to which wall a sensor is measuring by comparing the sensor's current orientation angle $\theta_p = \theta_k + \theta_{off}$ (where θ_{off} is the angular offset of the proximity sensor from the front of the robot) with the angles to the four corners of the field. The perpendicular distance d_{wall} from the robot's center to this wall is then calculated and converted to a sensor range R_p (along the sensor's axis) by

$$\widehat{R}_p = \frac{d_{wall}}{\cos(\theta_N)} - r_p$$

where θ_N is the angle between the axis of the sensor and the normal of the wall and r_p is the distance from the proximity sensors to the robot's center axis. For our proximity sensor calibration, we used $r_p = 3.7$ cm. We assumed the proximity sensors would output an ADC value that decreased exponential with distance and was proportional to a power of the cosine of the angle between the sensor axis and the wall normal, or

$$z = aR_p^b \cos^c(\theta_N)$$

where a, b and c are constants. We calibrated the proximity sensors over 754 measurements with various distances and angles, a subset of which is shown in Figure 3. Minimizing the sum of squared error between the measured and predicted ADC values, we found the following calibration constants:

$$\begin{aligned} a &= 1768.86 \\ b &= -2.22132 \\ c &= -1.62302 \\ \sigma_p &= 20 \end{aligned}$$

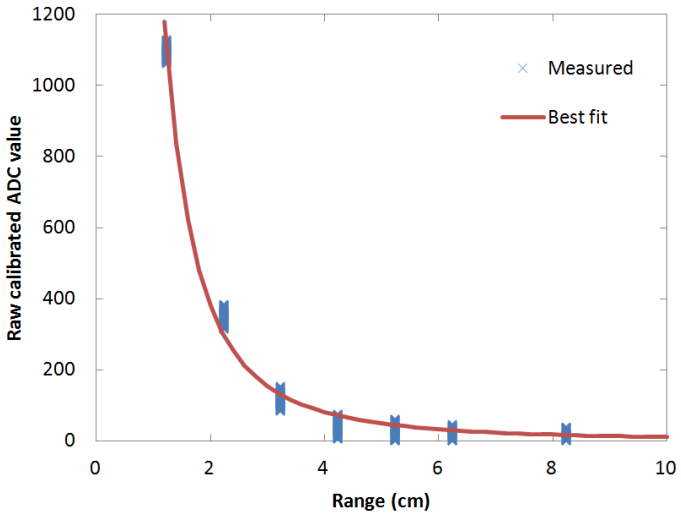


Figure 3: Proximity sensor calibration curve for $\theta_N = 0$

The scalar a reflects the sensitivity of the sensor. The exponent b is close to giving an inverse square law, which is expected–brightness should decrease with the square of distance. The cosine exponent c is negative, meaning that the response of the sensors increases as sensor axis departs from the wall normal and approaches the plane of the wall. A possible explanation for this seemingly unintuitive behavior is that the IR beam spreads and the brightness of the reflection is dominated by the portion that reflects off the portion of the wall closer to the robot than point on it that intersects the sensor axis.

For all cases, the EKF measurement model is

$$\hat{z} = a\widehat{R}_p^b \cos^c(\theta_N),$$

but the linearized matrix depends on the wall in view:

North wall:

$$H_1 = 0$$

$$H_2 = -\frac{ab\cos(\theta_N)^c d_{wall}^{b-1}}{\cos(\theta_N)}$$

$$H_3 = \frac{ab\cos^c(\theta_N)\sin(\theta_N)R_p^{b-1}d_{wall}}{\cos^2(\theta_N) - ac\cos^{c-1}(\theta_N)\sin(\theta_N)R_p^b}$$

South wall:

$$H_1 = 0$$
$$H_2 = \frac{ab\cos(\theta_N)^c d_{wall}^{b-1}}{\cos(\theta_N)}$$
$$H_3 = \frac{ab\cos^c(\theta_N)\sin(\theta_N)R_p^{b-1}d_{wall}}{\cos^2(\theta_N) - ac\cos^{c-1}(\theta_N)\sin(\theta_N)R_p^b}$$

East wall:

$$H_1 = -\frac{ab\cos(\theta_N)^c d_{wall}^{b-1}}{\cos(\theta_N)}$$
$$H_2 = 0$$
$$H_3 = \frac{ab\cos^c(\theta_N)\sin(\theta_N)R_p^{b-1}d_{wall}}{\cos^2(\theta_N) - ac\cos^{c-1}(\theta_N)\sin(\theta_N)R_p^b}$$

West wall:

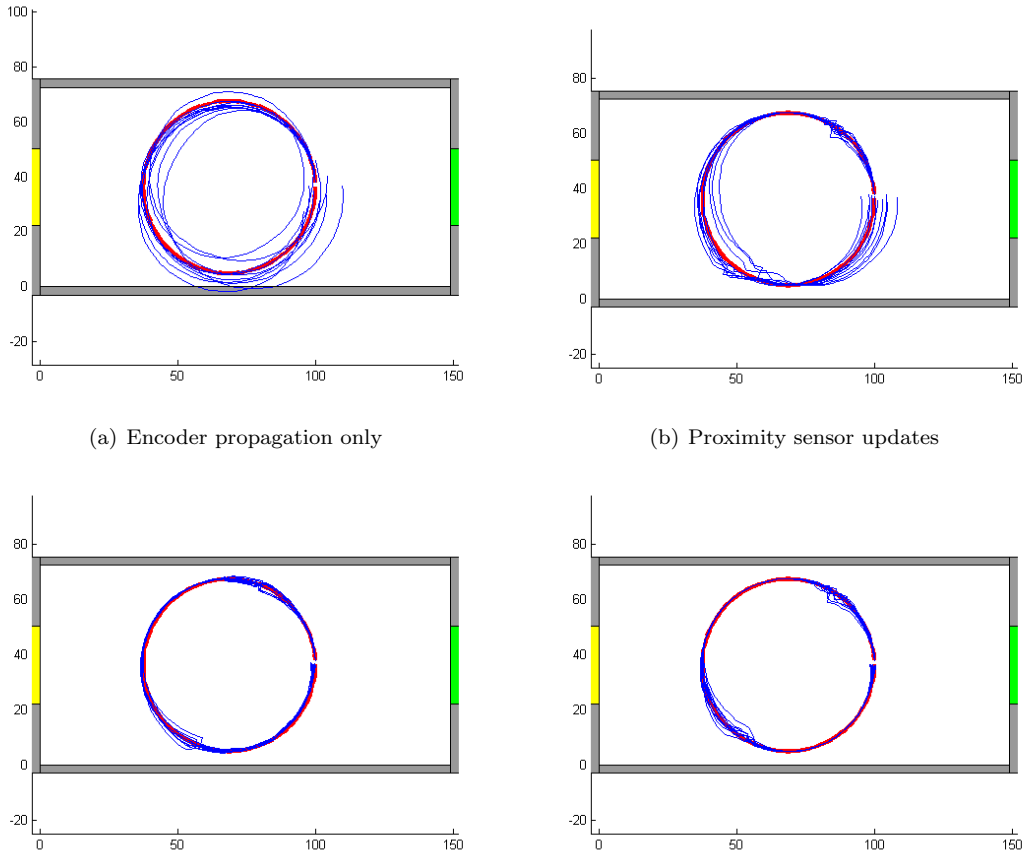
$$H_1 = \frac{ab\cos(\theta_N)^c d_{wall}^{b-1}}{\cos(\theta_N)}$$
$$H_2 = 0$$
$$H_3 = \frac{ab\cos^c(\theta_N)\sin(\theta_N)R_p^{b-1}d_{wall}}{\cos^2(\theta_N) - ac\cos^{c-1}(\theta_N)\sin(\theta_N)R_p^b}$$

In a multi-robot scenario, measurements cannot always be trusted. Other robots might obscure goalposts, moving the observed yellow or green edge, or drive near the robot, obscuring the wall from the robot's proximity sensors. To combat this, we implemented χ^2 outlier rejection to reject measurement updates that are not consistent with the robot's pose estimate. If the normalized error

$$\chi^2 = \widetilde{\mathbf{z}}_k \mathbf{S}_k^{-1} \widetilde{\mathbf{z}}_k$$

corresponds to a measurement likelihood less than 5%, the measurement is rejected. If a proximity measurement is rejected and the ADC value is significantly above the noise floor, the sensor is most likely detecting another robot. In this case, the filter returns the estimated distance to the robot $\hat{d}_{robot} = h^{-1}(z_k) = (z_k/a)^{1/c}$ to the strategy algorithm, which may take appropriate action.

Simulated EKF performance is shown in Figure 4. The sensors complement each other because each is only to make useful observations for a subset of the trajectory. The proximity sensors only decrease variance significantly when the robot is near the walls, the camera can only make updates when the goal is in view, and the encoder propagation carries the "best guess" of the robot's pose when the sensors are not providing information. The filter thus allows the robot to make strategic decisions without active observation—a robot with food is able to drive it towards a goal, even if the goal cannot be seen.



(a) Encoder propagation only

(b) Proximity sensor updates

(c) Camera updates

(d) Proximity and camera updates

Figure 4: Simulated EKF performance. The red circle represents the true path taken by the robot (counterclockwise from right) and the blue lines represent EKF-estimated trajectories. Using only encoders (a), the variance grows monotonically with distance traveled. Using the proximity sensors to detect the walls (b), the vertical variance is minimized when the robot is close to the north/south walls, while the horizontal variance grows monotonically. Using the camera to detect goalposts, the variance remains low whenever a goalpost is in view. Trajectories estimated using both proximity and camera updates (d) have the lowest total variance.

3 Simulation

The goal of our simulation task was to model as closely as reasonably possible our robot implementation, and to thus use simulation results as an inspiration or direct contribution to the development of our on-robot strategies.

To this end, we defined a reasonable fascimile of the tournament arena using MASON, and additionally developed a genetic programming implementation to run repeated simulation trials in order to evolve superior strategies.

3.1 Definition

Our simulation setup, using the MASON Java library, attempts to replicate at a high level the setup used in the foraging milestones. The competition field is based on the dimensions of a standard soccer field, 150 units wide by 220 units long, and is implemented as a Continuous2D space in MASON. Each e-puck is represented by an oriented circle 12 units across. Food "pucks" are 4 units across, and the goals are 75 units wide, or half the field width. Note that the simulation assumes image coordinate space, so the "width" of the field is actually the height, with y positive down, while the "length" of the field is the width, with x positive right. Robot rotation is positive clockwise, the opposite of the actual e-puck robots.

The food is distributed randomly around the field based on coordinates generated using Java's Mersenne Twister implementation. Any food position that intersects with already-placed food, a robot, or is too close to a wall is rejected and retried. We used the seed 1333593072282, which happens to be the Unix epoch in milliseconds at the time we first started running simulations (on April 5th). Using a consistent PRNG seed from run to run guaranteed the same behavior if we maintained the same conditions, which is useful for both debugging and reproducibility.

In terms of sensors and actuators, we model only the IR range sensors and the camera, plus the motors. We do not model any structure of the robots (i.e. there are no wheels, and the camera's focal point is the nose of the robot).

We track 16 range sensors, twice as many as in reality, evenly spaced on the robot's circumference. These sensors are "perfect", in that they are precisely report the distance of any robot or wall within their field of view. The range sensors on the actual robots report a reflection intensity, which falls off quickly after only a few inches, and they are very noisy. As discussed above, we found that sensor calibration varied non-trivially between individual robots.

The camera is implemented as a 30-pixel line, centered on the focal point. The simulated camera is a perfect projection of all objects in view, ignoring walls. Other robots can obscure food or the goal, but we assume perfect segmentation by allowing the robot to only see the unobscured portions of the frontmost object of interest.

In earlier implementations, we did not consider object collisions, even though in the actual tournament robots regularly got stuck against walls, shoved around food pucks, or get entangled with one another. We found that it was necessary

at least to prevent robots from intersecting eachother, and to allow robots to move food objects out of the way. "Picking up" food is not modeled with a kicker similar to the real robots, but rather we just check that food is close enough and in front of the robot when a pickup is attempted.

Robot motor speed is expressed in field units per step, and like the real robots, the simulated ones have differential steering. The speed maximum is 0.8 units (in forward or reverse). The wheels are assumed to be on the robot's circumference, on the exact left and right of the robot. Thus the midpoint of the robot moves at the average speed of the two motors. After the initial move, the robot's position is adjusted for collisions with the wall or other robots, and then shoves any food that might be nearby. Once the final position is determined, the position of any food being carried is updated so as to remain exactly in front of the robot.

The main loop of the simulation is shown in pseudocode in Algorithm 3. We run the robots on each team separately in turn, and then check for any food that was successfully delivered to each team's goal. When simulating a particular robot, its sensor state is updated as described above by calling UPDATECAMERA and UPDATERANGES, then its loaded strategy is executed with EVAL to update the state and motor speeds, then the movement is handled as described above in MOVE. All robots start in the `search` state. Each team's goal uses SCORE to clear any food in range (simulating a person picking up food out of the real tournament arena) and to update the running score.

Algorithm 3 Update simulation each time step

```
procedure STEP(sim)
    for team ∈ sim.teams do                          ▷ Two teams
        for robot ∈ team.robots do                   ▷ Three robots
            c ← UPDATECAMERA(robot, sim)
            r ← UPDATERANGES(robot, sim)
            state, speeds ← EVAL(state, c, r)
            MOVE(speeds, sim)
        end for
        score ← SCORE(score, team.goal, sim)
    end for
end procedure
```

3.2 Grammar

Our original implementation of the EVAL call from Algorithm 3 was written in Java and directly manipulated the state of the robot object, checking states and using hard-coded logic to decide what speed the motors should be set to. In order to enable evolution of strategies, we implemented a domain-specific language using list-style S-expressions. Each node consists of an operator atom followed by zero or more children, depending on the operator. Children can be either atoms or nested lists, again depending on the definition of the operator.

Operators are separated into two categories by return type: those which return a numeric value (treated internally as a `double`), and those which return a true/false value. Operators taking numeric arguments may restrict them to integers or reals as appropriate. The list of operators by type and argument count is shown in Table 2; the top-most operator must be `step`. During evolution we ensure that crossover and mutation do not replace an operator with one from the other category, as described below.

While some of the operators are just used for logic, the remainder effectively call a method on the invoking robot. `getMidpoint` and `getWidth` return integer pixel values from the camera. `getTravel` and `getRotations` get distances based on the movement of the motors. `getRange` gets the distance reported by a specific sensor, and `pickUp` and `drop` handle food carrying. `inState` and `setState` take one of four possible state flags, and `setSpeed` sets specific motor values, thus determining the robot's movement.

# Arguments	Logical	Numeric
0	`drop`, `noop`, `pickUp`	`getMidpoint`, `getRotations`, `getTravel`, `getWidth`, `vnoop`
1	`inState`, `not`, `setState`	`getRange`
2	`eq`, `gt`, `gte`, `lt`, `lte`, `setSpeed`	
3	`if`	
N	`and`, `or`, `step`	

Table 2: Grammar operators by type and argument count

An example is shown in Step Program 1, which is the baseline used for the opposing team during GP evolution, and for both teams during early testing. This program was manually tweaked several times over the course of development in order to provide good comparison performance.

The full Extended Backus-Naur Form (EBNF) specification of this grammar is shown in Table 3. Our step program parser implemented in Java is a raw loop over characters, due to the excessive complexity of available parsing libraries such as ANTLR. The step program converter, which produces C code that is nominally compatible with the e-pucks, is implemented in Python using the pyparsing library [10].

3.3 Genetic Programming

Genetic programming requires the definition of three operations on population individuals: crossover, mutation, and fitness. In our GP approach, the population consists of step programs. To run a single generation, each individual is run against the baseline (Step Program 1) until all food is gathered or 20000 simulation time steps have been taken. We define the fitness in terms of the relative score as:

Step Program 1 Baseline step program used during testing and GP evolution

```
(step
  (if
    (inState search)
    (if
      (lt (getMidpoint) 15)
      (setSpeed -0.1 0.2)
      (if
        (gt (getMidpoint) 15)
        (setSpeed 0.2 -0.1)
        (if
          (lt (getWidth) 13)
          (setSpeed 0.2 0.2)
          (and (setSpeed 0.0 0.0) (pickUp))))))
  (if
    (inState backup)
    (if
      (and (gt (getTravel) -12) (gt (getRange 8) 2))
      (setSpeed -0.2 -0.2)
      (and (setSpeed 0.0 0.0) (setState uturn))))
  (if
    (inState uturn)
    (if
      (lt (getRotations) 0.5)
      (setSpeed 0.2 -0.2)
      (and (setSpeed 0.0 0.0) (setState search))))
  (if
    (inState carry)
    (and
      (or
        (if
          (lt (getMidpoint) 8)
          (setSpeed -0.1 0.2))
        (if
          (gt (getMidpoint) 22)
          (setSpeed 0.2 -0.1))
        (if
          (or (gt (getRange 0) 6) (lt (getWidth) 15))
          (setSpeed 0.2 0.2)
          (and (setSpeed 0.0 0.0) (drop))))
      (or
        (if (and (lt (getRange 2) 4) (lt (getRange 14) 4)) (setSpeed -0.2 -0.2))
        (if (lt (getRange 0) 4) (setSpeed -0.2 -0.3))
        (if (lt (getRange 4) 4) (setSpeed -0.1 0.2))
        (if (lt (getRange 3) 4) (setSpeed -0.1 0.225))
        (if (lt (getRange 2) 4) (setSpeed -0.1 0.25))
        (if (lt (getRange 1) 4) (setSpeed -0.1 0.275))
        (if (lt (getRange 15) 4) (setSpeed 0.275 -0.1))
        (if (lt (getRange 14) 4) (setSpeed 0.25 -0.1))
        (if (lt (getRange 13) 4) (setSpeed 0.225 -0.1))
        (if (lt (getRange 12) 4) (setSpeed 0.2 -0.1))))
    (or
      (if (and (lt (getRange 2) 6) (lt (getRange 14) 6)) (setSpeed -0.2 -0.2))
      (if (lt (getRange 0) 6) (setSpeed -0.2 -0.3))
      (if (lt (getRange 4) 6) (setSpeed -0.1 0.2))
      (if (lt (getRange 3) 6) (setSpeed -0.1 0.225))
      (if (lt (getRange 2) 6) (setSpeed -0.1 0.25))
      (if (lt (getRange 1) 6) (setSpeed -0.1 0.275))
      (if (lt (getRange 15) 6) (setSpeed 0.275 -0.1))
      (if (lt (getRange 14) 6) (setSpeed 0.25 -0.1))
      (if (lt (getRange 13) 6) (setSpeed 0.225 -0.1))
      (if (lt (getRange 12) 6) (setSpeed 0.2 -0.1)))))
```

step	=	left, 'step', [{logical}], right ;
left	=	[space], '(', [space] ;
right	=	[space], ')', [space] ;
space	=	{? any whitespace ?} ;
logical	=	no-op | food logical | n-ary logical | if | not | comparison | state logical | speed ;
no-op	=	left, 'noop', right ;
food logical	=	left, 'drop' | 'pickUp', right ;
n-ary logical	=	left, 'and' | 'or', [{logical}], right ;
if	=	left, 'if', logical, logical, [logical], right ;
not	=	left, 'not', logical, right ;
comparison	=	left, 'eq' | 'gt' | 'gte' | 'lt' | lte', numeric, numeric, right ;
state logical	=	left, 'inState' | 'setState', state, right ;
speed	=	left, 'setSpeed', double, double, right ;
numeric	=	value no-op | sensor | range sensor | double ;
value no-op	=	left, 'vnoop', right ;
sensor	=	left, 'getMidpoint' | 'getRotations' | 'getTravel' | 'getWidth', right ;
range sensor	=	left, 'getRange', space, ? integer [0,15] ?, right ;
double	=	space, ? valid `double` ? ;

Table 3: Backus-Naur Form of our step program grammar

$$f = \frac{score_{us}}{steps} \frac{score_{us}}{score_{them}} \frac{1}{penalty}$$

This increases with strategies that collect food faster, as well as strategies that allow "our" team to outscore an opponent running the baseline strategy. A tie game in simulation would result in the fitness equalling the collection rate $\frac{score_{us}}{steps}$. The *penalty* was an additional factor we introduced late in development; most of the time it is set to 1 (no effect), but if for some reason an individual robot barely moves relative to its teammates, it increases to 10. This helps the GP to avoid converging on strategies that don't actively move. We could modify the application of this *penalty* factor to prevent the development of other undesirable strategies.

We define the mutation operation using a single global mutation rate m for each evolution run, expressed as a probability that any given node in a step program's tree will mutate in some way. The entire step program tree is walked from the root `step` node. The specific mutation that occurs depends on the grammar operator at that node; these are detailed in Table 4. Certain sensor-querying nodes are not capable of mutation, as it would change the step program too vastly with a single chance. The special `noop` and `vnoop` operators are generally not useful to include in a manually written program, as they have no effect, but we use them as "growth" nodes during mutation, with an inflated mutation rate of 0.5. They can thus quickly branch into novel step programs.

We define crossover between two individual step programs as selecting a random node from the first program, selecting a random node of a compatible

Operator	Mutation
`step`, `and`, `or`	Delete one random child
`if`	Rearrange alternative and consequent
`not`	Replace itself with its child
`eq`, `gt`, `gte`, `lt`, `lte`	Select a random comparator
`double` literal	Multiply by random $[0.5, 1.5]$ or if 0 set to $[-0.1, 0.1]$
`getRange`	Select a random range sensor $[0, 15]$
`inState`, `setState`	Select a random valid state
`setSpeed`	Modify both speeds as `double` literals
`noop`, `vnoop`	Select a random operator of matching type
`drop`, `pickUp`, `getMidpoint`, `getRotations`, `getTravel`, `getWidth`	No mutation

Table 4: Grammar mutation rules by operator

type from the second program (that is, they must both have the same return type), and then swapping them so that the first program's node's parent is now the parent of the second program's node, and vice-versa. To select a node at random, we walk the tree, accumulating all nodes of a given type (with no filter for the initial selection). A node is chosen uniformly from this flattened list.

The initial population of size s is seeded using one loaded step program that was manually written plus $s - 1$ mutations thereof. In earlier implementations, this seed was the same as the baseline program the algorithm is competing against.

The evolution process is given in pseudocode in Algorithm 4, with EVOLVE defined in Algorithm 5 and both described below. When simulating a particular individual, we can run the simulation multiple iterations i, incrementing the PRNG seed each time. This results in multiple initial food configurations, with the fitness of the individual being the average fitness across these iterations, and prevents an evolved strategy from being too sensitive to initial conditions.

Algorithm 4 The top-level genetic programming algorithm

```
procedure GP(baseline, seed)
    pop ← [seed]                                  ▷ Initialize population
    for i = 1 → s do
        pop ← pop + MUTATE(seed)
    end for
    fittest ← null
    for g = 0 → N do                              ▷ Run N generations
        pop, fittest ← EVOLVE(pop, baseline)
    end for
    return fittest
end procedure
```

Algorithm 5 A single GP evolution generation

```
procedure EVOLVE(pop, baseline)
    for ind ∈ pop do                                          ▷ s individuals
        ind.fitness ← SIMULATE(ind, baseline)
    end for
    f_total ← 0                                               ▷ Get fitness stats
    f_max ← 0
    fittest ← null
    for ind ∈ pop do
        f_total ← f_total + ind.fitness
    end for
    for ind ∈ pop do
        if ind.fitness > f_max then                           ▷ Update fittest individual
            f_max ← ind.fitness
            fittest ← ind
        end if
        f_rand ← RANDOM · f_total
        f_sum ← 0
        parents ← []
        for parent ∈ pop do                                   ▷ Select parents
            if f_sum ≤ f_rand & f_rand < f_sum + parent.fitness then
                parents ← parents + parent
            end if
            f_sum ← f_sum + parent.fitness
        end for
    end for
    pop ← [fittest, MUTATE(fittest)]                          ▷ Breed s − 2 offspring
    for i = 2 → s do
        children ← CROSSOVERANDMUTATE(parents[i], parents[i + 1])
        pop ← pop + children
        i ← i + 2
    end for
    return pop, fittest
end procedure
```

After running the simulation for all of the individuals in the population at a given generation, we select parents for the next generation using a variation of the roulette method. The total fitness of the population is calculated, and each individual is assigned a range proportional to its fitness relative to the total fitness. We maintain the same population size between generations, so for each parent slot s a random number in the range $[0, fitness_{total}$ is generated to select the individual corresponding to that fitness range. An usually fit individual for its generation could be selected to be a parent multiple times in this manner.

Each parent was crossoved over pairwise with one adjacent parent, producing

two offspring; all offspring were then additionally mutated. The population was then replaced with these new children and evolution continued on to the next generation. The output of the system was the fittest individual in the population after N generations. In later versions of the system, to avoid generations with a significant reduction in fitness, we propgated the fittest individual, and one mutation thereof, into the next generation, along with $s - 2$ children selected and crossed over as before.

We extended this original GP algorithm to allow for co-evolution of multiple strategies; that is, each individual in the population was now a set of step programs, up to one for each robot on the team. This also required extending the simulation to accept multiple strategy inputs for a team. The intention was to evolve distinct roles for each robot in the overarching team strategy. In order to keep the average population fitness up during evolution, only a single step program member of a given parent was selected at random for crossover with its counterpart at the same index in the other parent. All constitunt step programs of both descendents were then mutated, as in the single-strategy case.

4 Results

4.1 Competition

A single robot using the strategy described prevously with our tuning parameters could capture about 8 food items in 4 minutes of gameplay. The single robot performance can be considered a benchmark, such that successful multi-robot collectives should perform better. In our milestone 1 competition, our robots retrieved 13 of 15 food items in 4 minutes. The collective was therefore more successful than a single robot by 5 food items. It is important to note that one of the three robots was not functioning well and did not contribute any food items. In milestone 2 we competed in two matches against opposing teams. The first match was relatively successful with 8 food items collected by each team resulting in a clean arena in about 2 minutes. The second match was less successful with 3 food items collected by our team and 6 food items collected by the opposing team in 2 minutes. Although the code was unchanged between the two matches, the proximity sensors seemed to be tripped more easily in the second match, resulting in the robots sometimes backing away from food or the goal.

4.2 Simulation

We performed a wide variety of experiments using our genetic programming and simulation system. While the primary goal was to produce strategies that would be usable on the e-puck robots, we wanted to generally flex the capabilities of our implementation. Our early experimental runs, as described in our progress presentation, were primarily "shakedown" runs to exercise the system and find problems.

These initial trials are how we discovered the necessity of positional constraints (the GP learned a strategy that was more fit simply by "cheating", turning off the range sensors and driving robots through one another). Similarly, the *penalty* factor to the fitness f was added after we found that our "gatherer" seed strategy was consistently evolving into a static "goalie", which tended to ward off opponents as shown in Figure 5.

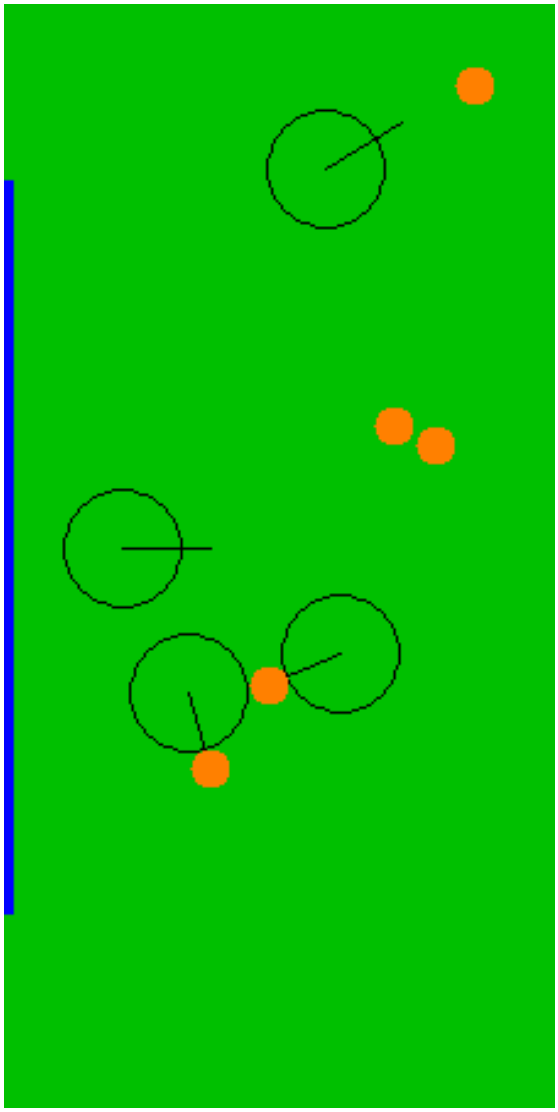


Figure 5: Evolved stationary goalie

Selecting reasonable simulation parameters that will result in successfully

evolved strategies is difficult, especially with such a large search space. We performed some short-duration experiments to help determine reasonable ranges for some of these parameters, including the population size, number of generations, and mutation rate.

We would expect that a very low mutation rate would not allow for sufficient exploration by the GP algorithm, and that a very high mutation rate would result mostly in completely non-functional individuals (that is, their step programs would instruct the simulated robot to not move, or be unable to see food, or similar). Since we propagate the fittest individual across generations, an overly mutated population would tend to preserve the seed individual while constantly churning dead mutant children. In Table 5 (graphed in Figure 6), you can see that a low mutation rate produces fitter output individuals and that eventually the mutations are so detrimental that only the initial baseline individual has non-zero fitness. These runs were performed with a single simulation iteration ($i = 1$) on a population of size $s = 20$ for $N = 5$ generations.

Mutation Rate	Average Fitness	Maximum Fitness
0.0	0.001449	0.003051
0.01	0.000501	0.003020
0.02	0.000516	0.002556
0.03	0.000241	0.002376
0.04	0.000070	0.001250
0.05	0.000155	0.002933
0.06	0.000147	0.001832
0.07	0.000067	0.001250
0.1	0.000063	0.001250
0.2	0.000063	0.001250

Table 5: Fitness with increasing mutation rate. Parameters $s = 20, N = 5, i = 1$

Evolutionary algorithms in general benefit from large populations, as this allows them to more exhaustively explore the problem space (with a diverse population) while also being more robust to individual die-off. This is particularly important in our GP, where mutations and crossover are more likely to produce an indvidual with fitness 0 than not, by producing a step program that while syntactically valid produces degenerate robot behavior. To test this, we varied s while holding m constant at 0.01 based on that result while performing $i = 1$ iterations for $N = 5$ generations. The trend for increasing population is shown in Table 6 and Figure 7. The only downside of increasing the population size is the corresponding linear increase in run time.

Because the fittest individual is propagated unmodified to the next generation, the maximum fitness of the population increases monotonically. The average fitness can fluctuate, especially if there are a large number of individuals in the population which have low or zero fitness even as the best individuals improve. This can be seen in the longitudinal plot over $N = 20$ generations shown in Table 7 and Figure 8, using parameters $s = 40$, $m = 0.01$, and $i = 1$. This

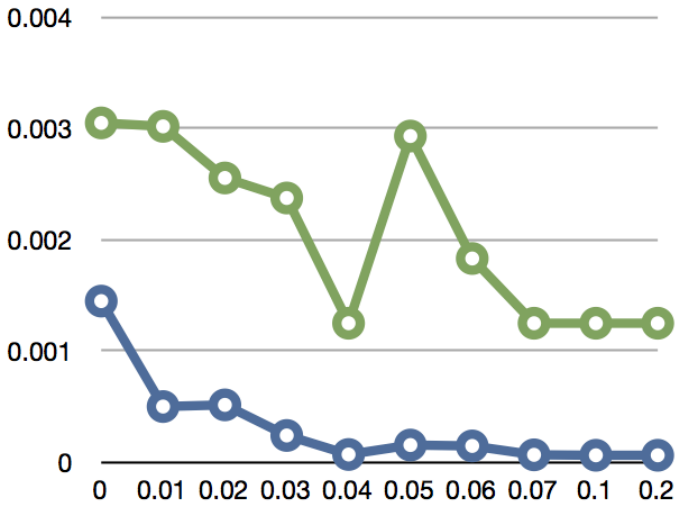


Figure 6: Maximum (green) and average (blue) fitness vs. increasing mutation rate. Parameters $s = 20, N = 5, i = 1$

Population Size	Average Fitness	Max Fitness
4	0.000436	0.001262
10	0.000730	0.002109
20	0.000501	0.003020
30	0.000867	0.002975
40	0.000540	0.002975
60	0.000756	0.003132
80	0.000660	0.003388
160	0.000749	0.004611

Table 6: Fitness with increasing population size. Parameters $m = 0.01, N = 5, i = 1$

particular individual appears to have learned to "bounce" when dropping food at the goal, backing up quickly and then moving forward again before turning back to the rest of the field. It's increased fitness appears to be due mostly to making faster turns.

We ran various other configurations to view the resulting strategies in the simulator UI. One individual, which started from the baseline seed plus a gatherer seed, and was run with parameters $s = 20, m = 0.01, N = 10, i = 2$ had fitness 0.002493, which by chance was reached in the initial round of mutation. Its behavior was an uninteresting optimization of the seed strategies, depsite having roughly four times as many operations in its output step program.

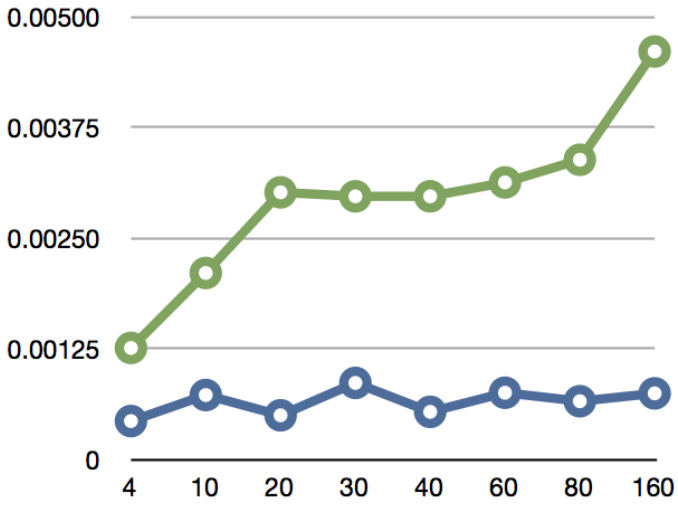


Figure 7: Maximum (green) and average (blue) fitness vs. increasing population size. Parameters $m = 0.01, N = 5, i = 1$

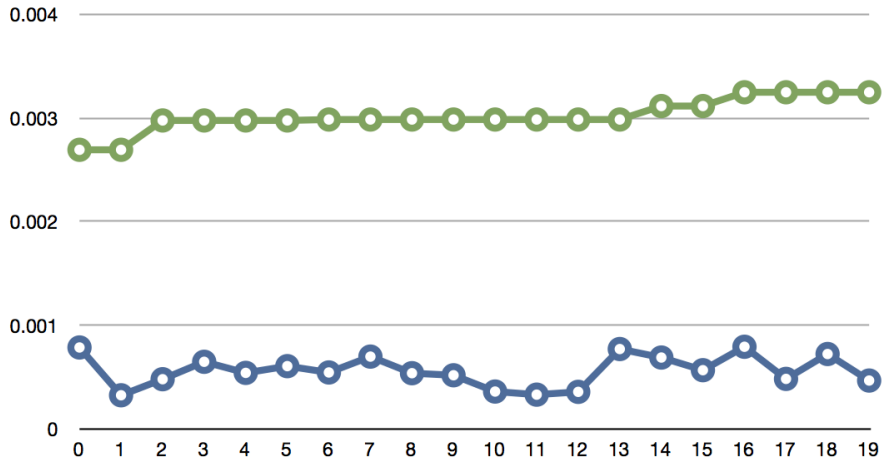


Figure 8: Maximum (green) and average (blue) fitness over time. Parameters $m = 0.01, s = 40, i = 1$

5 Discussion

The grammar implementation is a definite success, allowing for more flexible programming of the simulator. However, without a functional C converter

Generation	Average Fitness	Max Fitness
0	0.000785	0.002692
1	0.000324	0.002692
2	0.000479	0.002975
3	0.000647	0.002975
4	0.000540	0.002975
5	0.000605	0.002975
6	0.000544	0.002984
7	0.000697	0.002984
8	0.000537	0.002984
9	0.000517	0.002984
10	0.000360	0.002984
11	0.000330	0.002984
12	0.000357	0.002984
13	0.000770	0.002984
14	0.000688	0.003114
15	0.000566	0.003114
16	0.000792	0.003247
17	0.000482	0.003247
18	0.000722	0.003247
19	0.000466	0.003247

Table 7: Fitness over time. Parameters $m = 0.01, s = 40, i = 1$

(which requires more unification of function interfaces between our robot and simulator code), it unfortunately has little bearing on the actual robot task.

The genetic programming, while working, does not seem to improve significantly on the performance of the manually created step programs. For the most part it just learns to be more reckless by increasing the values of arguments to `setSpeed` and decreasing the sensitivity to the return values of calls to `getRange`, while simultaneously injecting a lot of extraneous operations into the S-expression tree which are never called. This implies that the fitness function could be improved to generate more interesting behavior, and to allow more step programs to have non-zero fitness.

One of the most interesting GP results came in the processing of introducing the *penalty* factor to the fitness function. The impetus was that we didn't want to waste a robot sitting in front of the goal, so we should penalize any individual that contained a step program that barely moved. In a follow-up run after introducing this change, the system learned to have that "goalie" robot circle in front of the goal, thus meeting the movement requirement while still achieving the performance of partially blocking the opposing team!

We had also experimented briefly with running starting from seeds that contained only no-ops, but without a significant change to the fitness function that provides some non-zero fitness even if $score_{us}$ is 0, the population has

nothing to work with. Perhaps a very large population and multiple rounds of "growth" mutation could generate nearly-random programs that converge on useful functionality?

5.1 Future Work

The simulation and genetic programming method could be improved in a few key ways, beyond simple parameter tuning. The GP fitness function, for example, operates only at the team level, which means that an individual improvement could be rejected if another team member is underperforming. The simulation does not model the robot's kicker at all, and in fact assumes a perfect grip on food. In the real-world arena, we observed several interactions not handled in simulation, such as robot entanglement and carrying multiple food pucks.

The only strategies we were able to successfully optimize were the baseline searcher, a gatherer, and in an unexpected development, a goalie. Some kind of sweeper role, where the robot systematically shoves food around, might have interesting benefits. This would require finishing the implementation of Kalman Filter-based localization, and implementing an EKF in the simulator. This would also allow us to simulate more realistic sensors by introducing a noise model.

Another area of continuing work would be to complete our converter from step programs to C code and attempt to run evolved programs on the robots. This would allow us to use the relative speed of the GP runs to test a wide variety of strategies without the slow cycle time of loading flashing new programs onto the e-pucks and running them in the lab.

5.2 Contributions

Nick Ward developed the GP simulation and obtained the results presented. Andrew Reiter and Pierre-Emile Duhamel worked on the e-puck implementations for competitions. Andrew Reiter developed the EKF implementation.

References

[1] Kok Seng Chong and Chong Lindsay Kleeman. Accurate odometry and error modelling for a mobile robot. In *In IEEE International Conference on Robotics & Automation*, pages 2783–2788, 1997.

[2] L.N. De Castro. *Fundamentals of Natural Computing: Basic Concepts, Algorithms, And Applications*, chapter 3. Chapman & Hall/CRC Computer and Information Science Series. Chapman & Hall/CRC, 2006.

[3] John H. Holland and Judith S. Reitman. Cognitive systems based on adaptive algorithms. *SIGART Bull.*, (63):49–49, June 1977.

[4] John R. Koza. Genetic programming: a paradigm for genetically breeding populations of computer programs to solve problems. Technical report, Stanford, CA, USA, 1990.

[5] Sean Luke. *Multiagent Simulation and the MASON Library*. George Mason University, August 2011.

[6] Sean Luke, Claudio Cioffi-Revilla, Liviu Panait, Keith Sullivan, and Gabriel Balan. Mason: A multiagent simulation environment. *Simulation*, 81(7):517–527, July 2005.

[7] Sean Luke, Charles Hohn, Jonathan Farris, Gary Jackson, and James Hendler. Co-evolving soccer softbot team coordination with genetic programming. In Hiroaki Kitano, editor, *RoboCup-97: Robot Soccer World Cup I*, volume 1395 of *Lecture Notes in Computer Science*, pages 398–411. Springer Berlin Heidelberg, 1998.

[8] Bruce A. Maxwell, Nicolas Ward, and Frederick Heckel. A human-robot interface for urban search and rescue. In William D. Smart, Bill Smart, and Magdalena D. Bugajska, editors, *AAAI Mobile Robot Competition*, volume WS-03-01 of *AAAI Technical Report*, pages 7–12. AAAI Press, 2003.

[9] John McCarthy. Recursive functions of symbolic expressions and their computation by machine, part i. *Commun. ACM*, 3(4):184–195, April 1960.

[10] Paul McGuire. *Getting started with pyparsing*. O'Reilly, first edition, 2007.

[11] Francesco Mondada, Michael Bonani, Xavier Raemy, James Pugh, Christopher Cianci, Adam Klaptocz, Stphane Magnenat, Jean christophe Zufferey, Dario Floreano, and Alcherio Martinoli. The e-puck, a robot designed for education in engineering. In *In Proceedings of the 9th Conference on Autonomous Robot Systems and Competitions*, pages 59–65, 2009.