IT3708 Assignment 1 | A Controller for Swarm Behavior in Webots

Kristoffer Hagen & Mads Wilthil, spring of 2014

Part A: Implementing the Proposed System

System Description

Our controller is written in python and attached to this report. We have not used the .c files given, but taken inspiration from these in writing our own behavioral models. Details in our model is in comments within the controller itself, while this report will focus on the overall description.

Basing our system on Brooks Subsumption Architecture, we have named our three sub-behaviors "Survive", "Retrieve" and "Stagnation". An overview of the system is shown in Figure 1.

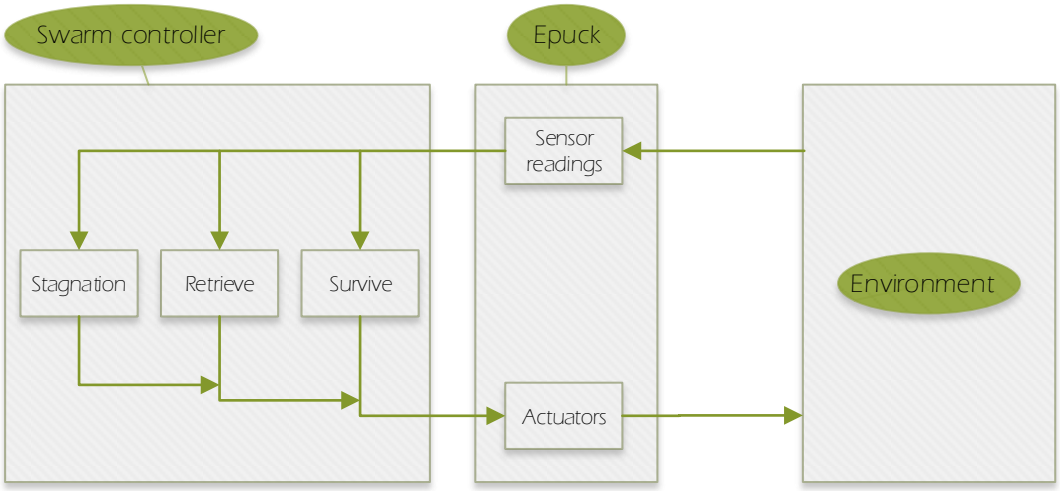

FIGURE 1: BROOKS SUBSUMPTION ARCHITECTURE FOR OUR EPUCK SWARM CONTROLLER

Survive is the most basic of the modules, and is heavily based on random movements. The Epuck move about in a random manner, by setting both its wheels to a random velocity. Should it collide or get close to obstacles, it will steer away or simply back up should it be cornered. The IR-sensors are used to avoid walls and other Epucks, while the camera is used to search for the food source. If the food source is located, the controller will let the Retrieve-behavior take precedence over Survive, and the Epuck will converge towards the box. In order to locate the box we take a picture with the Epuck camera after every action and look for the color of the box in five separate pixels. This is shown in Figure 2.

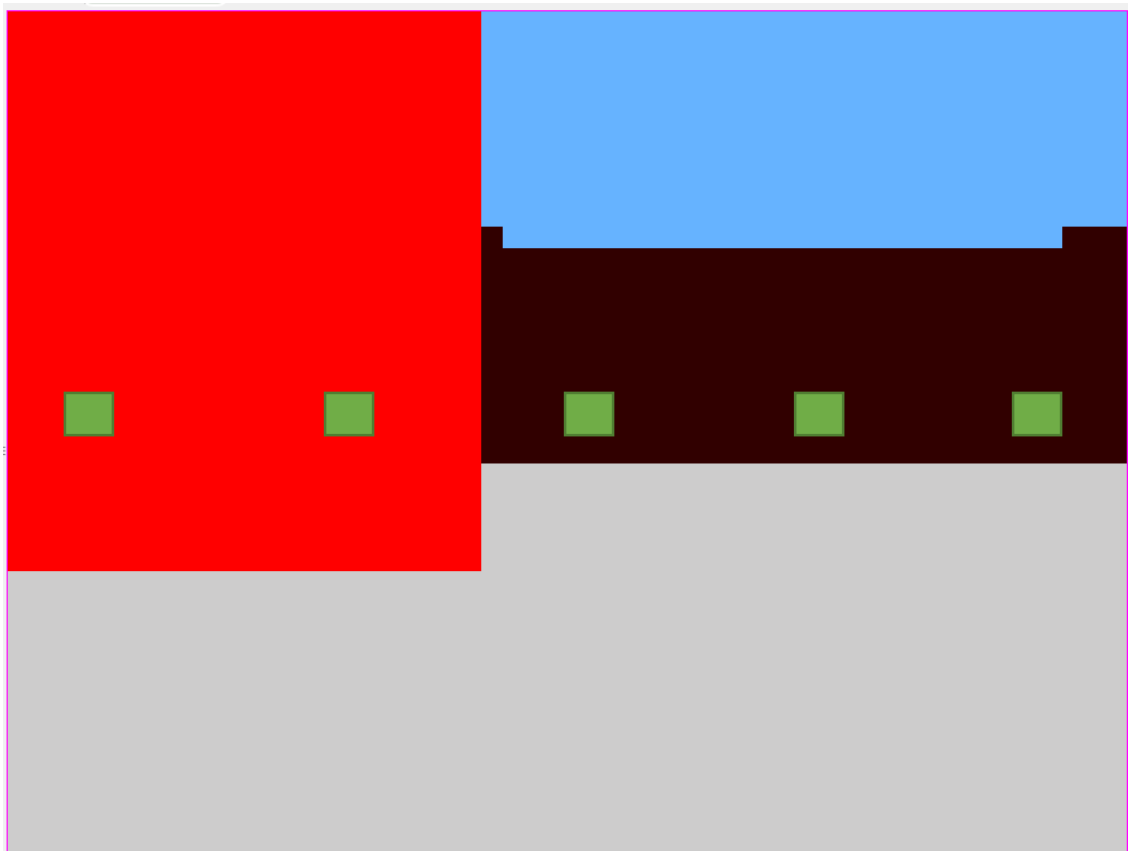

FIGURE 2: THE EPUCK CAMERA, WITH THE AREAS WE SEARCH HIGHLIGHTED IN GREEN (APPROXIMATIONS ONLY)

Retrieve can – roughly – be divided into two sub-modules. Getting to the box, and pushing the box. When an Epuck discovers the box, it will head towards it and find an available spot among the side it can see. If other Epucks are already pushing the box, it will go beside them and push. Do note that this is not done to reward pushing among other Epucks; Stagnation takes care of that. It is simply done to have the Epuck prefer pushing the box itself, not other Epucks. Once at the box the Epuck will attempt to push it, attempting to align itself towards the box. The closer the robots are to a box, the more accurately will it position itself to apply a normal force on the box surface. This is most useful if an Epuck positions itself between two other Epucks, to push directly on the box and not its comrades. This is illustrated in Figure 3.

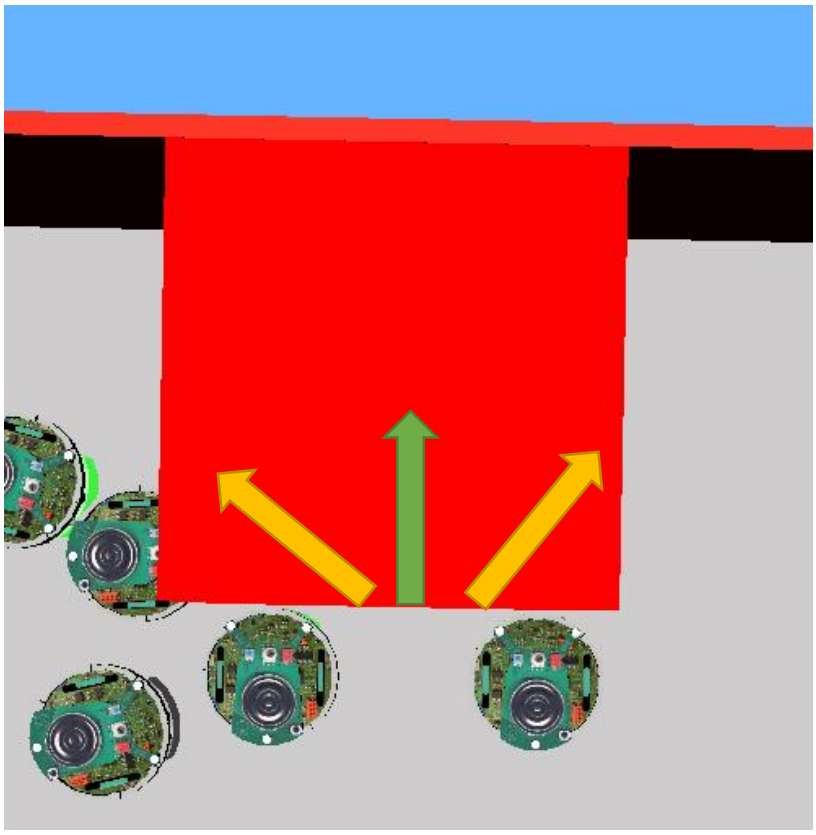

FIGURE 3: THE INTENTION OF WRIGGLE IS TO APPLY FORCE IN THE DIRECTION OF THE GREEN ARROW, AS PUSHING THE OTHER EPUCKS (AMONG THE YELLOW ARROWS) ARE A WASTE OF ENERGY

Stagnation takes care of repositioning the Epuck if it is unsuccessful in pushing the box. Each robot has a confidence value, which states how pleased the robot is in its current position. Being close to other friends will increase this value, while pushing alone for a long time will decrease it. The Epuck will attempt to wriggle for at bit, to apply force at different angles, before it decides to reposition. To reposition, the robot simply backs off, choses left or right at random, and moves to the corresponding side of the box to attempt another push. Once the Epuck is at the correct side, Retrieve will handle the repositioning of the Epuck; it is, after all, what the module was designed to do.

Expected and Unexpected Behaviors

An expected behavior rose from our work in making the robots choose available spots when approaching a box. If several Epucks were engaged in pushing a box, and one decided to quit, any incoming robots will take its place quite quickly. This is shown in Figure 4.

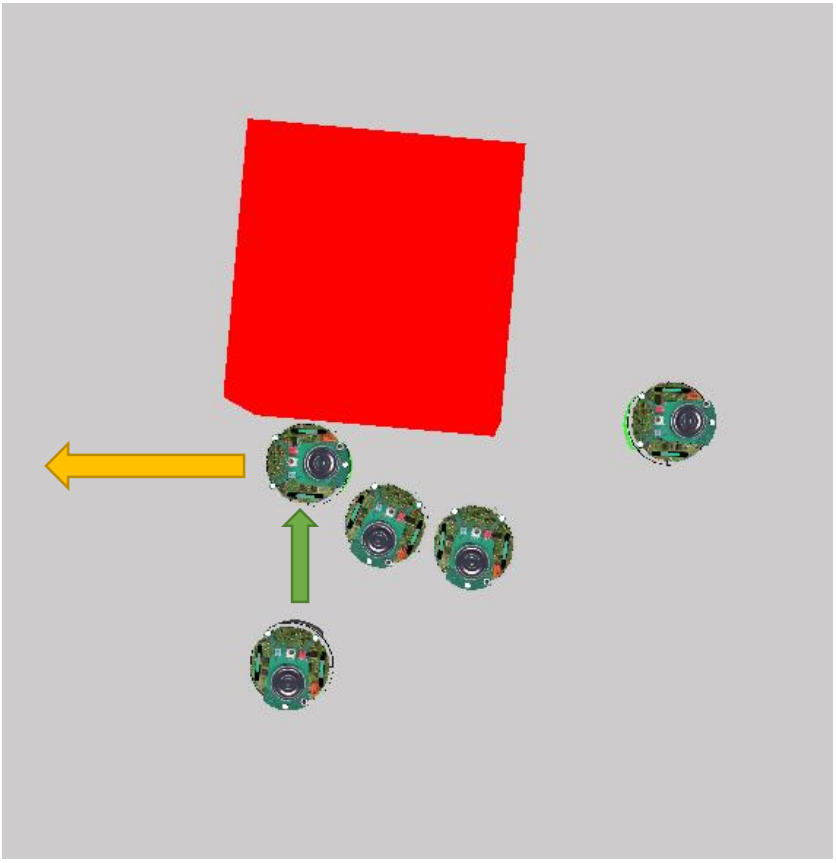

FIGURE 4: EXPECTED BEHAVIOR - AN EPUCK LEAVING THE BOX (YELLOW ARROW), WITH ANOTHER APPROACHING TO TAKE ITS PLACE (GREEN ARROW)

An unexpected behavior we found was that a collection of 3-4 Epucks would create a **too strong** group, and push towards each other for as long as we had the patience to watch. This is shown in Figure 5. At the longest, this went on for as much as 10 minutes. To fix it, we had to improve Stagnation to increase the chance of backing off, even with many companions surrounding you. Another issue was that a bots pushing the box would bump into another bot that was also pushing the box at the same side, and then believe that the other bot was also the box, leading to incorrect realigning.

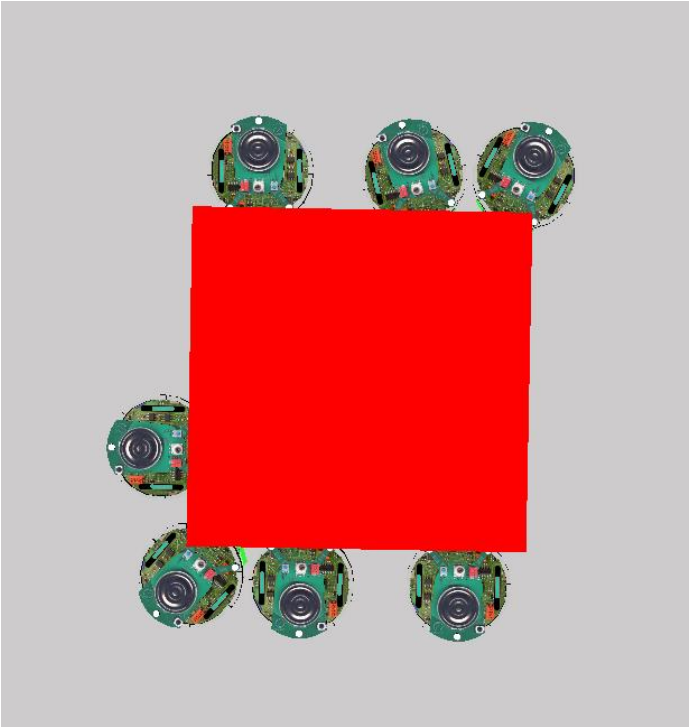

FIGURE 5: UNEXPECTED BEHAVIOR - TWO GROUPS OF EPUCKS PUSHING TOWARDS EACH OTHER, REFUSING TO GIVE UP

PART B: IDENTIFYING WEAKNESSES AND IMPLEMENTING IMPROVEMENTS

SUGGESTED IMPROVEMENTS

Initially we had some trouble getting the bots to push the box in a constructive manner after it was spotted and approached. A step we implemented to improve this was for the bot to attempt to keep the difference between the two forward most sensors as minimal as possible while pushing, this let the bots continue to push perpendicular to the box even when the box is moving on a separate axis. To solve the above problem, regarding groups of bots pushing in opposite directions, we made use of the Epucks accelerometer. If a robot did not move at all for some time, it gets a higher chance to turn away and search again. This will stop Epucks from pushing even if the object does not move. It took some time tweaking this mechanism, as the robots must still be able to wait for a group to form. We wound up using a threshold for the number of turns a robot could stand still; after that, it would reset its search with a 5 % chance. The details in using the accelerometer is covered in the controller code.

SYSTEM PERFORMANCE

To test our improved architecture, we ran 10 simulations and timed each run. A run taking longer than 10 minutes was marked as a failed run. The average time for the box to reach the wall was 2 minutes and 40 seconds, with a best time of 51 seconds. We had to quit simulations 2 times due to timeout, giving our controller a success rate of 83 %. An interesting note is that if a run was successful, it was resolved in less than 5 minutes.

PART C: ADVANCED TASKS

TWO-BOX SOLUTION

In order for the Epucks to split up into two groups, we used different colored boxes: the original red, as well as a blue box. To preserve the solution in the case of one box, we implemented an initial survey method so that every bot scanned the visible area and added all different kinds of food to a list, then it randomly selected one of the foods and ignored the other .If only a red box is present, the Epucks will converge to that. If both a red and a blue box is present, they will converge to a random box. One weakness with this is that Epucks does not always divide evenly into teams. Sometimes they will split into teams of 5 and 2, or even 6 and 1. This leaves us with a method that is not foolproof in the two-box case, but will still usually manage to push the boxes to the walls.

Our two-box world is shown in Figure 6. We decreased the mass of the boxes to 0.25, and also decreased the box side size to 0.15. This meant only two Epucks were required to push the box, but only two Epucks would fit among any side. Our code should handle any increasing amount of bots without any problem, though this was not tested.

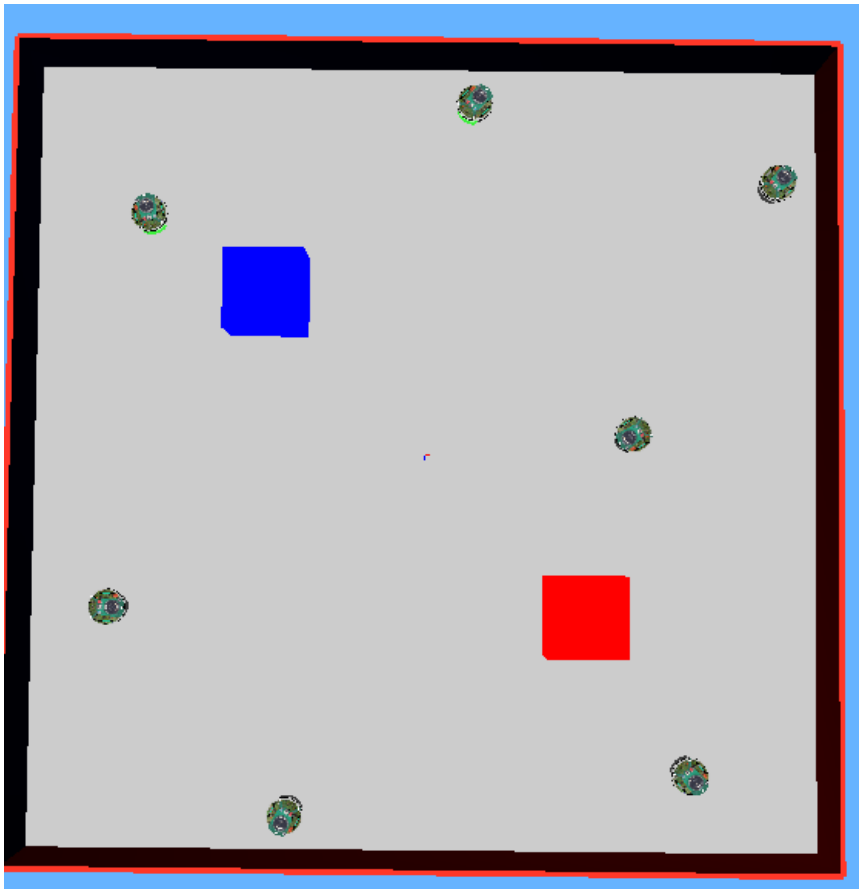

FIGURE 6: THE TWO-BOX SCENARIO, WITH TWO DIFFERENT FOOD SOURCES

SYSTEM PERFORMANCE

We approached performance testing in the two-box scenario in the same way we did with a single box. The average time was 5 minutes and 5 seconds, with a best time of 2:16. We had to abort the simulation 3 times in our testing, leaving us with a success rate of 77 %.