Homework Module:

A Controller for Swarm Behaviour in Webots

Purpose: Understand swarm behaviour by implementing a controller for box pushing task in Webots.

1 Assignment

In this project you will design a controller based on Brooks Architecture [2] to explore cooperative transport of a box in Webots environment. Intended box pushing task draws inspiration from swarm behaviour of ants observed often during food retrieval. The idea is to create a decentralized system invoking group behavior through simple mechanisms which if successful, leads to an emergent self-organized behaviour.

To design controller for this task, it would be required to create a box and eight e-pucks inside an area surrounded by walls (for dimensions and physics of the box and environment , see [1]) in the Webots environment. At the beginning, all e-pucks are randomly distributed in the environment and box is kept in the middle. Then as system is turned on, e-pucks have to locate the box, converge to it, and collectively retrieve it back to the hive (any of the four surrounding walls). E-pucks can interact with their environment using proximity and IR sensors, and left and right wheels are used as actuators.

To get more insight about box pushing task in webots, read thesis **Swarm Intelligence in Bio-inspired Robotics** [1].

In a more advanced task e-pucks have to retrieve two food boxes. They will therefore have to divide in two teams. This team division aspect will have to be done autonomously by the robots without global information. *Hint* : you can rely for example on the color leds and the camera of the robot.

2 Reminder

2.1 Swarm Behaviours

Many species of animals self-organize into coherent groups called swarm, flocks, schools, etc. These coherent groups enhance to fitness of individuals within it in a variety of ways including moving towards desirable goals or avoiding threats etc.. Remarkably, the group behavior does not involve any central control. Local interaction with a simple set of rules can lead to remarkably complex behaviors.

Typical approaches to controlling artificial systems relies on top-down control where one describes desired qualities and finds solutions by optimization. Unfortunately these approaches seldom scale well, therefore controlling large number of artificial objects can be rather challenging.

An alternative is to seek biologically inspired solutions such as reactive systems approach where agents do not plan every action they perform, instead action is a response to what happens in the environment. In this approach, agents always maintain interaction with their environment, and every input has a given output therefore, it doesn't need a lot of memory or processing power. It doesn't matter how big or complex the environment is, since the input size will stay the same and the actions are static.

2.2 Brooks Architecture

A behavior-based approach resembles much of what we could find in a reactive system and some would even label it as a reactive subsystem. It was made famous by Rodney Brooks and his subsumption architecture [2], and gained popularity in the field of robotics because it allowed for real-time processing systems that could work in complex environments.

A behavior is in most cases described as a sequence of interactions between e.g. an agent and its environment, where actions made by agent affect future perceptions and actions. In this system, behavior can be thought of as individual action selection function, where each behavior module can perceive and map inputs to a task-accomplishing action. Brooks claimed that intelligent behavior was created by a direct link between perceiving and acting (reactive) and that no reasoning was needed .

In 1986 Brooks proposed a behavioral decomposition of robot control [2] – see Figure 1, different from that of the functional. With a behavior-based architecture, one can decompose the problem into task-achieving actions and get direct access to actuators and sensors, which would make the system able to operate in parallel to other behaviors. It needs to directly experience and sense its environment hands on, and act upon it so forth. Each layer in the new architecture is placed on top of each other and illustrates a stack of parallel behaviors. The higher levels don't use the lower level as subroutines, but a set of pre-existing competences. This is called the subsumption architecture.

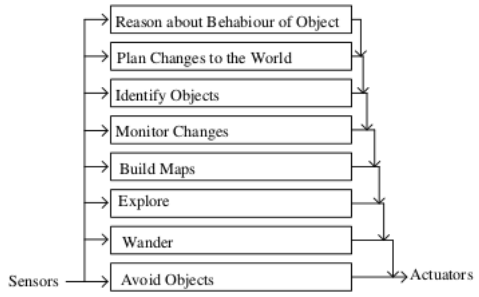

Figure 1: Brooks Subsumption Architecture

In the process of making this system one should design, test, redefine until the layer works to the users satisfaction before moving on to next layer. When more layers are added, the system can take use of the competence from all. The lower, more basic layers should deal with the survival of the system, while the higher should be design to concentrate on goal achieving behaviors. The higher layer has priority over the lower layers, meaning that the higher layers can suppress a lower layers behavior. It is important to do an incremental design, so the higher layers won't be triggered unnecessary to ruin the survival of the system.

3 Controller for Box Pushing Task

The overall Box Pushing system as shown in the Figure 2, is like a two-way dialogue between the behavior-based control system and e-pucks body. Body reactively sends the latest update about its environment with a question mark at the end, and controller responds with an answer. The e-puck sensory information needs to be collected and processed by control system, before it sends a command back to the actuators. The action of an e-puck continues to be performed until a new command presents itself next time step.

3.1 Behavior modules of Controller

As shown in the Figure 2, Box Pushing task behaviour is divided in three sub-behaviours described below

Search: This module contains behaviors on a proactive level. Even though ants seem to work all the time, they do rest, but what stimulates them to wake up and work? In this task it becomes a very low form of interaction to the environment, which is why the term proactive is more appropriate. The e-pucks need food to survive, so they are constantly looking for food to renew their energy, when

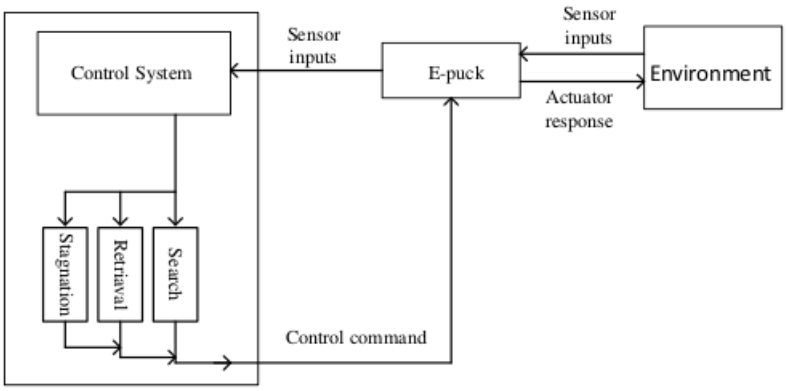

Figure 2: Brooks Subsumption Architecture

nothing else stimulates them to do otherwise. This search for food needs to be accommodated with an avoidance behavior to avoid obstacles and avoid becoming stuck.

Retrieval: It is a module made for behaviors invoking group behavior, which essentially emerges from robots sharing the same goal. The mechanisms are designed to trigger as a food source is detected. It needs to converge and align with the item before starting a push behavior to retrieve it.

Stagnation: It is a module made for behaviors which simply keep groups behavior progressing. At some part of the retrieving event, the item's motion may stop and progress yield as ant forces are applied from opposite directions which cancel one another. This can also occur when the group encounters obstacles. Whatever the case, e-pucks have to solve it the same way ants do – by spatial rearrangements. Through continuous ways of realigning and repositioning, the ants continue their progress towards the hive.

These behavior modules have been written in C and would be provided to you as support files. You can use them as such or can rewrite in the language of your choice, or can implement from scratch.

Note: For in depth description and implementation details of these modules, read chapter 6 of the thesis report [1].

Further, you need to implement a behavior-based control system in the language of your choice. This should be able to control artificial agents (e-pucks) to react from a direct link between perceiving and acting without need of any reasoning. Each behavior within behavior modules, shown in the Figure 2, follows this pattern of being triggered only when sensory input stimulates it to react.

With the higher behaviors subsuming the lower ones, the following points describe how the desired control system will operate:

- Stimulated by the mere fact of being awake and in working modus, a clear path lets the e-puck search for food.
- If any danger for collision appears on the path in front of the e-puck, it will turn either left or right to avoid
- Sensing IR light is like ants pheromones; it triggers e-pucks to converge, heading towards that direction.
- Upon arrival at the artificial food source a push behavior will trigger to retrieve the box.
- During the retrieving event stagnation may occur and e-pucks will try realigning with the box, creating a different angle of force.
- If stagnation is still imminent after realigning the last option is to reposition; finding a new spot to apply force.

4 Webots

The Webots software will soon be available as a floating license, but if this arrangement is not yet in place when you begin the assignment, simply go to the Webots homepage: http://www.cyberbotics.com/.

Then follow the menu path: Products/Webots/download to find the Demo version for your machine. Once the demo version is running, you can download a free PROfessional version for 30 days. You'll need that PRO version, since the Demo version won't allow you to compile new controllers.

Once Webots is installed, fire it up and go to the Help menu for access to the Introductory materials, a comprehensive User's Guide, and a good Reference Manual. In the Reference Manual, pay particular attention to Chapter 10, "Other APIs". There, you will find very useful overviews of the key classes and methods used in Webots. You will need to read a good deal of this documentation; all important details are not described in this homework-module description.

Webots scenarios are implemented as *worlds*, and each world can contain one or more robots, along with various stationary physical objects, such as walls and blocks. We will provide you with world, which you are free to use (as is) or modify. Most changes can be done using the "Scene Tree" window on the left side of the Webots window. Worlds are language independent, so regardless of your choice of language for programming the controller, this same world file should work fine.

When you load the given world into Webots, you will see a table top with an epuck robot and several colored blocks. Each of these components is defined in the Scene Tree. At the bottom of the Scene Tree is an EPUCK definition. Clicking on that definition shows its attributes. A new controller in Webots can be created using using the top menu item *Wizard* and choosing the "New Robot Controller" option. The wizard allows you to specify the programming language for your controller, with options such as C, C++, Java and Python. Most of the demo code that comes with Webots is written in C, C++ or JAVA. Python is a relatively new addition, so there are few demos. However, the base epuck-controller class that we provide, *epuck basic* is Python code. The functionality in epuck basic should translate easily to other languages; the method or function names are similar across the different API's.

You can define and edit your controller in the Webots window, or you can use your favorite editor and load it into Webots via the *revert* button in the controller-editor portion of the Webots window, which normally appears on the right. In our experience, the editor window provided by Webots is not that helpful for Python; its use of indentation is confusing. So you might opt to use your favorite editor instead.

4.1 File Structure

The normal practice in Webots is to define a project directory, such as *swarm-epuck*, inside of the Webots robots directory. It then contains 2 directories entitled *worlds* and *controllers*. Inside the former directory, simply place your e-puck.wbt file. However, inside the *controller* directory, you need to create another directory with the EXACT SAME NAME as your controller file (minus the file extension). So the file structure that you add into the Webots/projects/ directory should look something like this (where the robots directory should already exist):

```
robots
    swarm-epuck
        controllers
            my-swarm-controller
                epuck_basic.py
                my-swarm-controller.py
                any other supporting code for your controller
        worlds
            e-puck.wbt
```

The file extension on your controller file tells Webots the language of your controller, and it will then invoke the appropriate compilers, loaders, etc. to get your controller up and running.

4.1.1 Shell Variable Bindings in Webots

If you start up Webots by clicking on the ladybug icon, you will have the full power of languages such as C++, Matlab, Python, etc., but you will not necessarily have access to (your normal) bindings of shell variables such as PYTHONPATH. Without those bindings, Python in Webots will not know where to look for your files. It will look in the Webots sub-directory where your controller is defined (as shown above), but not, for example, in any of your normal directories (outside of the Webots file tree). This may or may not matter to you, as it is perfectly fine to simply include all of your code in a Webots directory (such as robots/swarm-epuck/controllers/my-swarm-controller/ above).

But if you'd like Webots Python to include your normal shell bindings, one simple approach is to start Webots from a terminal window. On the Mac, this is done by executing the following file:

Webots/webots.app/Contents/MacOS/webots

On Linux or Windows, look in similar locations for the proper executable. The bindings created by Webots will then be appended to those declared in your shell and you should be able to call all of your normal code during a Webots run.It is still important that the controller code resides in a directory with the same name as that of the controller file (i.e *my-swarm-controller.py* is in directory *my-swarm-controller* above).

4.2 Controller Modules and Classes

Webots provides the module named controller, which contains all of the robot-component classes such as Emitter and DistanceSensor. The controller module also defines the class Robot, which provides easy access to these components on simulated (or real physical) robots. Finally, controller contains the class DifferentialWheels, a subclass of Robot. This represents a special class of robots: those that have 2 wheels and use differential steering (i.e., the robot turns by applying different velocities to its wheels).

We provide you with the class EpuckBasic (in file epuck basic.py), which is a subclass of Differential-Wheels. Should you decide against using EpuckBasic in your controller, at least be sure to inherit from DifferentialWheels.

Note that although the code that we provide is written in Python, these same Webots classes (such as DifferentialWheels, Robot etc.) are available in C++, JAVA, and MATLAB versions of the Webots controller-building code.

4.2.1 The EpuckBasic Class

The file epuck basic.py contains the class EpuckBasic. Even if you do not code your controller in Python, you are wise to read through the code and comments in that file. EpuckBasic is built as an abstraction so that IT3708 students and other users can ignore a lot of the lower-level details of interfacing with Webots robots. If you use a language other than Python, you may want to define something similar to EpuckBasic in your language. Since most of the Webots classes are the same in all (supported) object-oriented languages, your C++ or JAVA versions of EpuckBasic will probably use many of the same calls as in the Python version.

1. **get proximities** - This returns a vector containing the values of the 8 distance sensors. These should all be integers between 0 and 4096.
2. **move, move wheels, and set wheel speeds** - Any (or all) of these can be used to set the wheel speeds of the robot. Normally, your code will set wheel speeds based on output values of motor-layer neurons, but the wheels will not be activated to run at those speeds until a call to do timed action occurs.
3. **run timestep** and **do timed action** - The former calls the latter, which sends the step(timestep) command to the robot, causing it to run for timestep milliseconds (of simulation time) with the current settings of the wheel speeds.

To force the robot to move, you thus have to do 2 things: a) set the wheel speeds, and b) call run timestep or do timed action. The timestep is actually stored as a property of the world. You can access and modify it by clicking on the "WorldInfo" attribute at the top of the Scene Tree in the Webots window. The WorldInfo attribute named basicTimeStep is an integer denoting the number of milliseconds in a simulated timestep. It has a default value of 32, meaning that anytime you call run timestep, the simulator will perform actions for 32 milliseconds of simulation time. How long this takes on your computer can vary - but in the world being simulated, it's 32 milliseconds worth of action.

The EpuckBasic class includes a basic setup method which loads the WorldInfo.basicTimeStep into it's own timestep slot. This method also instantiates the camera and distance sensors.

4.3 Epuck Details

Many of the details of the epuck robot, such as it's dimensions, sensor parameters, etc., or NOT easily available via the Webots window. To access these, go to the directory webots/projects/default/protos/ and then check out these text files (in separate subdirectories):

- sensors/EPuck DistanceSensor.proto - the specfications for the epuck sensors. To understand these, you'll need to check the Webots documentation.
- robots/EPuck.proto - the complete specification for the epuck robot. This references the DistanceSensor.proto specification, among others.

Many of the physical details of the epuck are already incorporated into the EpuckBasic class. For example, dimensions such as the axle length and wheel radius are used to calculate rotation angles during calls to the various spin methods. You should not need to worry too much about epuck specifications unless you choose to implement a very advanced task.

4.4 Webots Examples

Webots comes with many sample robots and worlds. Most can be found in the robots directory:

Applications/Webots/projects/robots

Support for Python is a very recent addition to Webots, so only a few Python examples come with Webots.

In the following directory:

Applications/Webots/projects/packages/python/controllers

you will find two directories, driver and slave, each of which contains a Webots controller written in Python.

4.5 Auxiliary Files

All three behavior modules (search.c, retrieval.c and stagnation.c) and corresponding header files would be provided as support files.

Note it that you would be required to include following header files in your control system to get access to e-puck robot and its hardware components.

- include webots/robot.h
- include webots/differential wheels.h
- include webots/distance sensor.h
- include webots/light sensor.h
- include webots/led.h

5 Deliverables

Part-a: Implement the proposed system

1. Describe briefly the overall system in text and diagrams. Document if you made any changes to the provided world. **3 points**
2. Implement control system for the box pushing task as provided in the assignment text and describe the expected and unexpected behaviours observed during simulation. **5 points**

Part-b: Identify weaknesses and implement improvements

1. Identify areas of the system that may be improved in the single box scenario. These areas may be in overall architecture of controller or at the individual sub-behaviour task. Provide suggestions (one or more) to address these improvements. **4 points**

2. Modify the given world or create a new world to implement these improvements. Show how well this new implementation performed during simulation. **4 points**

Part-c: Go toward more advanced tasks

1. Propose a new architecture to address the two boxes scenario. You will notably take care to explain how the new mechanisms don't disturb the resolution of the single box scenario. **2 points**
2. Modify the given world or create a new world to implement these improvements. Show how well this new implementation performed during simulation. **2 points**

- The report should be maximum 6 pages long, including diagrams.

6 Important Practical Issues: Please Read!!

- You can work alone or in groups of 2 or 3.
- This project will also be DEMONSTRATED for the instructor and course assistants and the demo will be arranged on 27th January, 2014. We do not expect perfect performance but your robots should be able to converge and recover from the stagnation while running simulation.

References

[1] Jannik Berg and Camilla Haukenes Karud. *Swarm intelligence in bio-inspired robotics*. PhD thesis, Norwegian University of Science and Technology, 2011.

[2] Rodney Brooks. A robust layered control system for a mobile robot. *Robotics and Automation, IEEE Journal of*, 2(1):14–23, 1986.