Q-learning 6
Samuel Rosas
Castor Köhler
Mikko Hemilä
Joni Tuppurainen

Project Documentation

15.12.2017

SCOPE

Our project has a great variety of functionality that should cover all the requirements of the topic, both minimal and optional. The main purpose of our program is to teach our robots to move forward. The progress of our robots is also visualized in real-time so you can see how they progressively learn increasingly effective ways of moving. The scope of this project can be divided into three main parts: the Q-learning algorithm, physics and graphics.

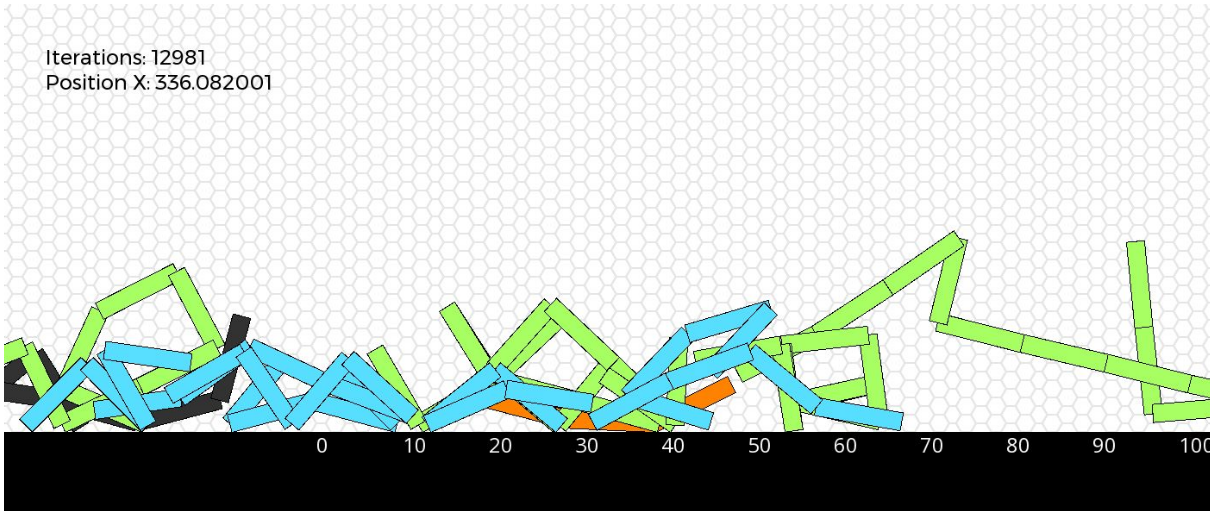
Q-algorithm:
Q[state][action] += alpha * (reward - punishment + gamma * next_state_max_q - Q[next_state])

Reward is the new x-coordinate minus the old x-coordinate. Punishment is a constant determined by user -- usually between 0 and 1 as 1 seems to be a good gain in x-coordinates for one movement within our simulation. Alpha is the learning rate (0 to 1), meaning how much a move alters the Q-matrix. Gamma is the discount factor (0 to 1), making future moves affect the equation less than the ones right ahead. Next_state_max_q gives the maximum Q value achievable by acting in the next state.

The Q Learning loop first makes sure the joints are in correct states, then takes an action according the largest achievable Q value from an action (or randomizes between them, with larger Q-values gaining more weight). After the joint has moved, it updates the Q-matrix with the Q Learning algorithm shown above.

The bots that use our Q-learning algorithms have been implemented with the game physics library Box2D. The bots are worm-like objects with a variable amount of joints (we've used three joints mainly). The bots capable of moving their joints freely, only restricted by their strength. The bots reside in a very basic world that consists of a floor, also implemented with Box2D.

In our program there are different kinds of robots, both robots that have individual q-matrices and so called "hive-minds" (or swarms as we call them) that use the same q-matrix and that way learn faster since they're all collaborating to the learning. These are depicted in different colors in the program so that you can easily see the difference in their progressions.



The above screenshot illustrates the different types of wurms (Genus: *Wurmus*) as we call them, that we have implemented.

The wurms are color coded as follows:
Orange (*Wurmus masterus*): Maister, the original wurm. First it was really shy and only wrote his position to us but as we gained his/hers trust it started to do more and more daring appearances. It updates its q-matrix independently and has been scientifically proven to be the BEST WURM EVER. Camera also follows this quality specimen even without vision.
Black (*Wurmus simplimus*): Independent bots that each update their own q-matrix.
Blue (*Wurmus collectivus*): Swarm-bots that share q-matrices.
Green (*Wurmus longimus*): “Goofy wurms” that try to move their joints in 90 degree angles, and have 7 joints. They also operate as a hivemind like the swarm-bots, but the results are more extreme and entertaining because of structure of the wurms.

The program has features that allow you to save the progression of the robots as well as load the progression from external files. The saving can be done manually but is also done automatically with a specified interval. The loading can also be done manually and is done automatically at the launch of the program. The Swarm-bots load and update their shared matrix every 100 actions they take. As this interval may happen at different times with the bots, their states within the world tend to differ greatly. Even more so as the time from their latest load furthers.

The drawing of the world and bots is done with SFML:s graphics library. We acquire the positions and rotations of the bots from the Box2D world and match their shapes with SFML shapes, that are then drawn onto the screen. The world also has a background, and waypoints that indicate how far the bots have traveled. SFML is also used to listen to keyboard presses, which we use to move the camera around in the world. Our program also contains the feature to fast-forward the learning of the robots. Furthermore, keyboard shortcuts can be used to change different parameters of the world, such as the torque of the of the robots’ joints etc.

PROGRAM ARCHITECTURE

The software comprises six classes and a main function. The classes are illustrated in the diagram below.

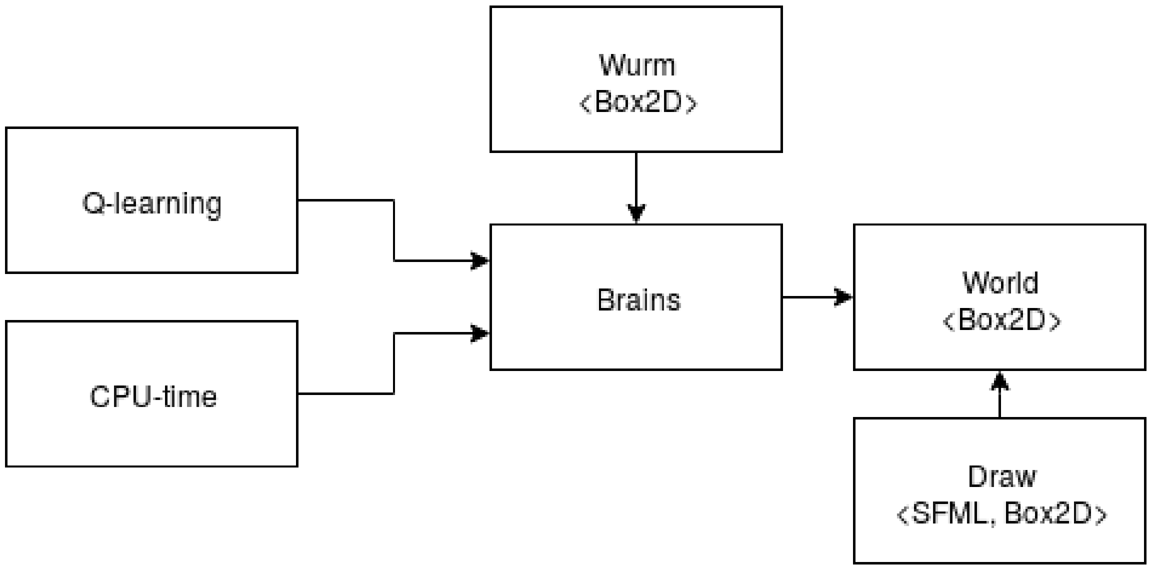


Image of the classes and their interfaces. The arrows mean that class pointed to uses the interface of the given class.

The **Wurm class** describes a single bot, defined as several Box2D bodies. The wurm has functions for setting and getting its position, joint angles, speed, etc. The wurm is nothing but an body with motorized joints, and needs a brain to function. Since the wurm is a Box2D entity, it also interacts with the world class, which describes the Box2D world that the wurms reside in.

The **Brain class** contains code that controls when calls to the q-learning algorithm are made. The class contains the Wurm and the Q-learning class and calls them both according to the conditions inside the Think() function. Calls to the q-learning algorithm are made based on the wurm’s joint angles, since the q-matrix can only contain certain positions of the joints (the positions have to be discrete, and few enough to be able to store in the matrix), the wurm has to be in the right position for the q-learning algorithm to be called. The brains are the interface through which we connect the q-learning algorithm and the wurm.

The **Q-learning class** contains all code that implements the q-learning algorithm and controls how the algorithm behaves. This class also implements the save and load functions of the program. It’s interface is accessed through the Brains class. It also stores many of the important values, such as joint amount, precision and name as it uses them the most often and thus has to provide Get functions for any other classes needing these values.

The **CPU-time class** is a class that is used to describe and monitor CPU time mostly while running the Q-algorithm. It is simply a class that includes the chrono tools to measure CPU time accurately with high resolution clock. Regardless of how overkill using a high resolution clock is, we wanted to get accurate runtimes and this proved to be the most precise solution.

The **World class** describes a Box2D world. Parameters such as the friction of the world can be controller trough this class. The Box2D world inside the world class contains all the wurms in the program.

The **Draw class** is used to draw all the objects in the world with SFML. As can be seen from the diagram, it accesses both the Brains class and World class to draw everything in the program. Even though all the objects could be drawn through the World class, since all the wurms can also be found there, we found it more convenient to directly give the draw function brains, since it could use the info in the brains to draw the wurms in different colors depending on the type of brain the wurm has. To draw the shapes, Draw checks the Box2D shapes of the given objects and draws the corresponding SFML shapes.

The **main** function in the program contains instances World, Brains, and Draw. It creates new brains containing the physical wurms, that it adds to a world, and also creates a Draw class to draw all of it.

BUILDING AND USER GUIDE

Building

Requirements

The requirements for this program are only two libraries/programs: SFML and CMake. Box2D is also used, but we've included the library files as part of the project. Both SFML and CMake should be installed on Aalto Linux machines, but in case you're building our project on some other machine, you'll have to install these softwares as well. To install SFML on a Unix system you need to install the library *libsfml-dev* if it's not installed. Note that we're using SFML version 2.3, if you're using a newer version of SFML the compiler will throw some warnings, but the program will still compile. To install CMake you need to download it and follow the instructions on cmakes website *www.cmake.org*.

To build the program you need only need to do a few things. Make sure you have SFML and CMake installed.

1. Clone the git repository (or download and extract to a folder)
   ```
   git clone git@version.aalto.fi:ELEC-A7150/q-learning-6.git
   ```
2. Create a directory called *build* in the *src* directory.
   ```
   cd q-learning-6/src
   mkdir build
   cd build
   ```

3. In the *build* directory, run the command
   `cmake ..`
4. After you've done this you are ready to compile the program by running the command `make`.
5. The executable should now be located in *q-learning-6/src/build/qbot* and is called bot. You can run it from the *build* folder by typing `./qbot/bot`

Using the program

The program starts with asking following questions:
-Disable info & cpu_info (1 = yes, 0 = no)
-How many normal BLACK, default 0 (own Q-matrix [3][24]) do you want? Note: One is automatically created.
-How many swarm-wurms CYAN, default 5 (shared Q-matrix [3][24]) do you want?
-How many goofy_wurms GREEN, default 0, fun with high speed joints (shared Q-matrix [7][4]) do you want?

These allow the user some control over how many wurms are created, but even with all answers 0, one "Maister" wurm with orange color is created. It is advisable the first time the program is run that user answers 0 to all questions to see clearly what this wurm is.

When the program is running there are a large amount of keybindings that can be used to alter the programs parameters. Here is a list of the buttons and what they do:

KEY	**FUNCTION**
↑	Zoom In
↓	Zoom Out
→	Move Camera Right*
←	Move Camera Left*
Spacebar	Reset camera
H	Print help text in console
S	Save
L	Load
U	Fast-forward
R	More torque
F	Less torque

I	More Speed
K	Less Speed
E	More punishment**
D	Less punishment**
T	Increase simulation speed
G	Decrease simulation speed

* Camera always follows master wurm even if it is moved
** Punishment means reward or punishment based on the rate of movement

There isn’t really too much to add to the usage guide. Let the bots do their thing, you can fast-forward to speed up the learning process. To see “properly” moving wurm it is advisable that program is fast-forwarded multiple times (preferably upto 100 000 iterations, but smalled amounts probably also work). Tweak the parameters and have fun!

TESTING / QUALITY ASSURANCE

Throughout the project, we have strived to use a consistent code style to ensure that our code is similar throughout the whole program. Rule of five was followed but there was only one class which needed it: world and even this class just disallowed the copying and moving functions as there should only be one world. Other common programming practices that we’ve learned during the course were also followed as best as we could. Git has been used for version management, with branches and merge requests ensuring that the code is peer reviewed and no poor code is left in the master branch of the project. This should ensure that our code is of good quality and also serves as a form of manual testing, since the code will always be tried out by a reviewer before being merged to the master.

Testing was planned to be done through unit tests, but we didn’t have time to setup google testing so it was left. Instead we tested our code by printing out information in the console to ensure that the program run properly. Initially when the Box2D implementation was missing, Q-learning was tested with the same functions with the goal of resetting the joints to zero position from random starting point as efficiently as possible. This proved to be a good practice in examining how different alterations to the q-algorithm affected the speed of learning or the efficiency of the final solution the Q-Learning class would weight on using.

The proper function of the Q-Learning being essential for the program, a compilation of print commands was linked to PrintInfo function alongside CPU-time (for benchmarking), so calling PrintInfo not only would print all the specific values transformed inside all of the Q-Learning functions, but it would also tell the runtime of those functions, albeit altered by the print functions.

Code was also tested with valgrind and specially with the following command: “valgrind --leak-check=yes ./qbot/bot”. The command was run at build folder after compiling and running the program once. At the end we managed to create a program without any memory leaks. We also had a strange bug with valgrind raising a warning about

WORK LOG

Castor

Main responsibilities:

- Project documentation
- Graphics and UI
- Project structure and library linking

Week 46 (Project plan DL)

- 5h Learning about q-learning and setting up the basic structure for the project (file structure etc)
- 2h meeting with group
- 2h writing project plan

Week 47

- 7h Working on making CMake and our libraries work in our project
- 1h Meeting with group

Week 48 Midterm meeting

- 6h More work with CMake and finally getting it work
- 2h Catching up on implementation of our project
- 1h Meeting with group

Week 49

- 5h Working on the GUI

- Week 50 (Project demo, final week)

- 4h work on the camera
- 4h Project Documentation
- 1h Final Meeting

Samuel

Main responsibilities:

- CMake, getting project to compile, linking external libraries
- SFML, UI design, SFML and Box2D connection

Week 46 (Project plan DL)

- 2h meeting with group, planning project.
- 2h writing project plan.
- 5h learning about project materials, Q-learning, Box2D, SFML

Week 47

- 2h learning CMake
- 10h creating project structure, linking with CMake, trying to get external libraries to play nice with CMake

Week 48 (Midterm meeting)

- 5h struggling with CMake and Box2D
- 5h experimenting with Box2D
- 2h preparing for and participating in midterm meeting

Week 49

- 2h learning SFML
- 5h creating GUI for our program, combining Box2D and SFML, adding features to GUI
- 3h cleaning code

Week 50 (Project demo, final week)

- 7h Improving UI, fixing issues
- 2h preparing for project demo
- 1h project demo
- 5h write project plan

Joni

Main responsibilities:

- Q-learning expert
- Performance
- General tasks (useful functions, fixes, anything anywhere)

Week 46 (Project plan DL)

- 2h meeting with group, planning project.
- 2h writing project plan.
- 5h learning about project materials, Q-learning, Box2D, SFML
- 2h git ssh-key problems

Week 47

- 2h meeting with group
- 10h implementing Q Learning and any required functions

Week 48 (Midterm meeting)

- 10h helping with other tasks to get a functioning world asap
- 2h meeting with group

Week 49

- 4h cleaning code
- 5h performance testing
- 3h logs (printing to console in nice format with cpu-time)

Week 50 (Project demo, final week)

- 5h cleaning code
- 5h cleaning code
- 3h simulation altering keybindings
- 1h get references working
- 1h project demo
- 2h implement useful functions

Mikko

Main responsibilities:

- box2d (wurm and world), overall fixing, merging requests and small parts of project management
- also cleaning code and making it follow similar style

Week 46 (Project plan DL)

- 2h meeting with group, planning project.
- 2h writing project plan.
- 5h learning about project materials and tools
- 2h trying to get git accept ssh-key

Week 47

- 1h learning Box2d
- 8h trying to get box2d compiled, creating physical part of the wurm and world

Week 48 (Midterm meeting)

- 2h messing with box2d (Testbed and other kinds)
- 2h struggling with g++ to compile program with Box2D
- 4h making box2d joint to work between wurm body parts

Week 49

- 3h learning SFML and compiling a test program with it
- 6h merging all different main branches (sfml, box2d and q-learning) and making sure the program compiled.
- 4h cleaning code

Week 50 (Project demo, final week)

- 8h fixing issues and cleaning code
- 1h project demo
- 4h changing pointers to references and making sure valgrind can’t find any memory leaks
- 2h write project plan and fixing last bad pushes.