dynrules Documentation

Release 0.1.0

Marcus von Appen

May 22, 2013

CONTENTS

dynrules is a dynamic ruleset scripting package for Python. It uses the Dynamic Scripting technique to create adaptive AI scripts automatically from predefined rulesets.

Contents:

CHAPTER

ONE

INTRODUCTION

This is the documentation for the dynrules dynamic rules creation package. dynrules is a small package that allows you to create rule-based scripts for adaptive AI systems automatically. It uses the Dynamic Scripting technique and algorithms as described by Pieter Spronck in his paper *Adaptive Game AI with Dynamic Scripting*.

The focus of the package is on weighted ruleset management and script generation, rather than weight adjustments and fitness evaluation. There is only limited support for that kind of functionality and most of it has to be implemented by the specific user code.

dynrules is not meant to be a fully functional AI package, but rather a supportive system for creating your own adaptive AI quickly.

1.1 Prerequisites

You must have at least one of the following Python versions installed:

- Python 2.6+ (http://www.python.org)
- PyPy 1.8.0+ (http://www.pypy.org)
- IronPython 2.7.2.1+ (http://www.ironpython.net)

1.2 Installing dynrules

You can either use the python way of installing the package using distutils or the make command using the Makefile. Simply type

```
$ python setup.py install
```

for the traditional python way or

```
$ make install
```

for using the Makefile.

It must be said that the install target of the Makefile does not do anything different from the python way. It simply calls 'python setup.py install'.

1.2.1 Trying out

You also can test out dynrules without actually installing it. You just need to set up your `PYTHONPATH` to point to the location of the source distribution package. On Windows-based platforms, you might use something like

```
set PYTHONPATH=C:\path\to\dynrules\:$PYTHONPATH
```

to define the `PYTHONPATH` on a command shell. On Linux/Unix, use

```
export PYTHONPATH=/path/to/dynrules:$PYTHONPATH
```

for bourne shell compatibles or

```
setenv PYTHONPATH /path/to/dynrules:$PYTHONPATH
```

for C shell compatibles. You can omit the *:$PYTHONPATH'*, if you did not use it so far and if your environment settings do not define it.

Note: If you are using IronPython, use `IRONPYTHONPATH` instead of `PYTHONPATH`.

1.3 Binary packages

dynrules is not provided as binary package by the author. It might be that someone else set up such a package for your wanted operating system or distribution. Those packages are usually not supported by the author, which means that installation problems or similar issues, which do not target dynrules directly, should be escalated to the respective supplier of that package.

1.4 Notes on Mercurial usage

The Mercurial repository version of dynrules is not intended to be used in a production environment. Interfaces may change from one checkin to another, methods, classes or modules can be broken and so on. If you want more reliable code, please refer to the official releases.

CHAPTER

TWO

GETTING STARTED

dynrules uses a simple, but flexible class layout to configure rulesets and create scripts from them. In order to use the dynrules package, simply import it using

```
import dynrules
```

You now can start to create own rulesets and set up your script generator.

2.1 Creating rules - Rule

A rule defines certain criteria of an object's behaviour. The `Rule` class features the most basic needs to set up your own rules for scripts. It consists of an *id*, a *weight* and *code* that defines the rule's action.

Let's imagine, you have a game with a Warrior class that can walk in four directions and fight against enemies. Defining those actions might look like

```
class Warrior:
    def do_walk (self, direction):
        ...
    def fight (self, enemy):
        ...
    ...
```

Automating walking and fighting requires a lot of work regarding the priorities of when to walk and when to fight. Fighting a stronger enemy might be senseless, so the Warrior should walk away from it. Let's try to formulate some rules for it:

```
if is_enemy_at (direction):
    if warrior.strength > enemy.strength:
        warrior.fight (enemy)
    else:
        warrior.do_walk (~direction)
else:
    warrior.do_walk (direction)
```

What does the above code do? First it checks, whether there is an enemy at the specified direction the warrior should walk to. If it is, and it is weaker than the warrior, the warrior will fight it. Otherwise, the warrior will go into the opposite direction to escape a possible fight. At last, if no enemy is found at the given direction, the warrior will walk towards it.

This is a predictable, typical behaviour and sometimes it might happen that, although the warrior is weaker, it will fight the enemy with success. That means, for a more unpredictable behaviour, we have four possible actions:

1. warrior.strength > enemy.strength: fight enemy

2. warrior.strength < enemy.strength: flee from enemy
3. warrior.strength < enemy.strength: fight enemy
4. warrior.strength > enemy.strength: flee from enemy

The first and second rule are directly from the solution above. They make the most sense in those cases and should be preferred. Rule three and four shall offer some more unpredictable behaviour.

Let's formulate some rules for the enemy detection scenario using dynrules.

```
# Create a new Rule for fighting the enemy.
rule1 = Rule (1)
rule1.weight = 10
rule1.code = "if warrior.strength > enemy.strength: warrior.fight (enemy)"

# Create another Rule for fleeing.
rule2 = Rule (2)
rule2.weight = 10
rule2.code = "if warrior.strength < enemy.strength: warrior.do_walk (~direction)"

# Fighting a stronger enemy
rule3 = Rule (3)
rule3.weight = 5
rule3.code = "if warrior.strength < enemy.strength: warrior.fight (enemy)"

# Fleeing from a weaker enemy
rule4 = Rule (4)
rule4.weight = 5
rule4.code = "if warrior.strength > enemy.strength: warrior.do_walk (~direction)"
```

We set up the necessary rules. Now it's time to put them together in a `RuleSet` that takes care of them.

2.2 Weighting rules - RuleSet

Weighting a rule means to mark and measure its priority or importance within a set of applicable rules. The more important a rule is, the higher its weight should be. Measuring the importance of a rule is usually done by counting how often it is called. The result (successful or unsuccessful) often influences the rule's weight, but does not need to.

The `RuleSet` takes care of mesuring the importance and updating the weight of the `Rule` objects it contains. A `RuleSet` usually consists of applicable rules for a specific situation and lets you define methods for measuring and detecting the success of the rules. As the process of measuring the success solely depends on the surrounding system, the implemenation of that process can vary and the `RuleSet` class requires you to take care of it.

```
class MyOwnRuleSet (RuleSet):
    def calculate_adjustment (self, fitness):
        # Implement your own success detection here.
        pass

    def distribute_remainder (self, remainder):
        # Implement your own remainder method here.
        pass
```

`calculate_adjustment(self, fitness)` calculates the reward or penalty, each used rule receives. The *fitness* argument can be used to provide additional information, e.g. about the performance of the execution.

`distribute_remainder (self, remainder)` is called to distribute the difference between the total weight before and after the update once the weight updating within the `RuleSet` is done. This might be necessary to allow

a balancing of rule weights so that the total sum of all rules within a `RuleSet` will remain the same, for example. In reality however, such a distribution solely depends on the specific application needs.

The weight update process of the `RuleSet` looks like

```
def update_weights (self, fitness):
    # Initialise needed things.
    adjustment = self.calculate_adjustment (fitness)
    # Update rule weights with the adjustment and calculuate remainder
    self.distribute_remainder (remander)
    # Update new total weight.
    # return
```

To get a better idea about this, let's create a small `RuleSet` implementation for our previously created rules. We assume that the *fitness* we receive expresses the difference between the damage the warrior made and received during the execution of the rules.

```
class WarriorRuleSet (RuleSet):
    def calculate_adjustment (self, fitness):
        #
        # fitness = damage_warrior_caused - damage_warrior_received
        #
        # 1) a high fitness means, the warrior caused more damage
        # 2) a very low or negative fitness means, the warrior did not
        # cause that much damage or even received more than it caused.
        #
        # for case 1) we assume the execution of the rules to be
        # successful, for case 2) we do not.

        # We set the sucess/fail threshold to 3.
        if fitness > 3:
            # The execution was successful, the warrior is strong!
            # The adjustment will be the total fitness - threshold.
            return fitness - 3
        else:
            # The execution was not successful, the warrior is weak!
            if fitness < 0:
                # Lousy, simply return the negative fitness
                return fitness
            else:
                # Not so lousy, return a penalty value as difference
                # of threshold minus fitness.
                return - (3 - fitness)

    def distribute_remainder (self, remainder):
        #
        # Here we distribute the difference of the last total weights
        # and newly calculated total weights.
        # Give each rule the same fraction.
        #
        count = len (self.rules)
        if count == 0:
            return # Safety net, if no rules are there.

        fraction = remainder / float (count)
        for rule in self.rules:
            rule.weight += fraction
```

Now we can add the created rules from above.

```
warriorruleset = WarriorRuleSet (0, 20)
warriorruleset.add (rule1)
warriorruleset.add (rule2)
warriorruleset.add (rule3)
warriorruleset.add (rule4)
```

The both arguments of the constructor, *minweight* and *maxweight* are the boundary limits for rules contained in a `RuleSet`. They define the upper and lower weight limit, each rule can have.

From now on, the `WarriorRuleSet` is fully functional and can update rule weights as necessary.

To add another level of automation and to create scripts from the rules, a `LearnSystem` will be necessary however.

2.3 Generating scripts - LearnSystem

The `LearnSystem` class is used to create scripts automatically from an existing `RuleSet`. It generates the scripts in a programming language neutral manner which means, that it only uses the *code* attribute of `Rule` objects for creating the output.

Additionally the `LearnSystem` can add code to be executed before and after the rules are entered to make the generated script fully functional for the specific task and environment. The creation of a script thus consists of the following tasks.

1. Create script header
2. Select rules and create code
3. Create script footer

To create a `LearnSystem` for the `WarriorRuleSet`, only a single line of code is necessary.

```
warriorlearnsystem = LearnSystem (warriorruleset)
```

The `LearnSystem` is now full functional and you can start generating scripts for the warrior.

```
warriorlearnsystem.create_script ("scriptfile.scr", 4)
```

`create_script` will create a new script, insert a header, add rules to it and then add the footer. You can specify the maximum amount of rules to be added by setting the second argument to the required value.

```
# Add a maximum of 10 rules.
warriorlearnsystem.create_script ("scriptfile.scr", 10)
# Add a maximum of 3 rules.
warriorlearnsystem.create_script ("scriptfile2.scr", 3)
```

You can modify several attributes and methods of the `LearnSystem` to tweak it to your personal needs.

`create_header()` and `create_footer()` are used to create necesary code to add before and after the rules. That can be initialisation and finalisation code, checks or whatever is necessary for the target system. Both methods return a string containing the code to add.

```
class OwnLearnSystem (LearnSystem):
    def create_header (self):
        # Create header code
        return 'def execute_rules (object):\n' + \
               '    selected_rule = None\n'

    def create_footer (self):
        # Create footer code
        return '    return selected_rule\n'
```

The above class would generate the following code:

```
def execute_rules (object):
    selected_rule = None
    #
    # RULE CODE
    #
    return selected_rule
```

The *maxscriptsize* attribute allows you to define the maximum size in bytes of a script to generate. *maxscriptsize* does not take the header and footer into account, but only the code generated from the rules.

```
# Limit the size of the code generated from the rules to 4 kB.
warriorlearnsystem.maxscriptsize = 4096
```

maxtries limits the rule selection process, so that it does not take infinite trials to find a rule to add. This can be very helpful to limit the time spent on selecting rules.

```
# Only try to find new rules 50 times.
warriorlearnsystem.maxtries = 50
```

That's it. We now have a basic dynamic scripting system that can select rules, create scripts and update the rule weights upon execution of the scripts. Now it is time to integrate all of it into the AI logic code!

CHAPTER

THREE

PYTHON API REFERENCE

class Rule (*id : object*)

Creates a new `Rule` object with the given id.

`Rule` is a simple class type that carries a weight indicator and arbitrary code data for usage in the dynamic script generation process.

code

Gets or sets the code of the `Rule`.

id

Gets the id of the `Rule`.

used

Indicates whether the `Rule` was used or not.

weight

Gets or sets the weight of the `Rule`.

class RuleSet (*minweight : float*, *maxweight : float*)

Creates a new, empty `RuleSet`.

`RuleSet` is a rule container class that manages rules, their weights and the weight distribution for the rules. The *minweight* and *maxweight* parameters are the minimum and maximum weight boundaries, each rule's weight has to stay in.

maxweight

Gets or sets the maximum weight to use for rules.

minweight

Gets or sets the minimum weight to use for rules.

rules

Gets the list of currently managed `Rule` objects.

weight

Gets the total weight of all managed `Rule` objects.

add (*rule : Rule*)

Adds a `Rule` to the `RuleSet`.

calculate_adjustment (*fitness : float*) → float

Calculates the reward or penalty, each of the activated rules recives. *fitness* hereby can be used as measure of the performance or whatever is suitable in the implementation.

This must be implemented by inheriting classes.

clear()
Removes all rules from the RuleSet.

distribute_remainder(*remainder : float*) → float
Distributes the remainder of the weight differences between the last weights and current weights.

The method must return a value. This must be implemented by inheriting classes.

find(*rid : float*) → Rule
Tries to find the Rule with the matching id and returns it. In case no Rule with the passed id exists, None is returned.

remove(*rule : Rule*)
Removes a Rule from the RuleSet.

update_weights(*fitness : float*)
Updates the weights of all contained rules.

Adapted from Pieter Spronck's algorithm as explained in Spronck et al: 2005, 'Adaptive Game AI with Dynamic Scripting'.

class RuleManager(*id : object*)
The RuleManager class takes care of loading and saving rules from arbitrary data sources. The base is an abstract class, which's load_rules() method must be implemented according to the specific needs of the application.

maxrules
Gets the maximum amount of rules to manage.

load_rules([*maxrules=-1*]) → [Rule, Rule ...]
Loads rules from the underlying data source and returns them as list. The *maxrules* argument defines the amount of rules to load. If it is smaller than 0, all existing rules should be returned.

This must be implemented by inheriting classes.

save_rules(*rules : iterable*)
Saves the passed rules to the underlying data source.

This must be implemented by inheriting classes.

save_rules_hint_file(*filename : string*, *learnsystem : LearnSystem*)
Saves a LearnSystem/RuleSet combination to a physical file.

class MMapRuleManager(*maxrules : int*)
A simple memory-mapped RuleManager implementation that does not load its rules from an external data source.

It is an extremely useful class for testing rules and basic algorithms, but due to the in-memory management of all rules, it should not be used in a productive environment, especially if large rule sets have to be managed.

By default, the MMapRuleManager class will reserve enough memory for the rules to manage, when it is constructed. It will **not** fill the rules with useful values though. It is up to caller to use load_rules() afterwards and fill the returned Rule instances with the necessary data.

maxrules
Gets the maximum amount of rules to manage.

load_rules([*maxrules=-1*]) → [Rule, Rule ...]
Returns the internally managed rules or a certain subset.

save_rules(*rules : iterable*)
This does nothing and will always return True.

save_rules_hint_file (*filename : string*, *learnsystem : LearnSystem*)
Saves a LearnSystem/RuleSet combination to a physical file.

class LearnSystem (*ruleset : RuleSet*)
Creates a new LearnSystem using a specific RuleSet.

The LearnSystem class takes care of creating new scripts based on a predefined RuleSet. It does not evaluate the scripts nor modifies the rules written to them.

The procedure of creating scripts is done using three phases:

•header creation

•rule code creation

•footer creation

The header and footer are freely choosable. You can simple override or reassign the create_header() and create_footer() methods to let them return your required code.

maxscriptsize
Gets or sets the maximum script size (in bytes) for inserting rules.

maxtries
Gets or sets the maximum amount of tries to insert a script rule.

ruleset
Gets or sets the RuleSet to use.

create_footer () → str
Creates the footer for the script file.

The default implementation does nothing.

create_header () → str
Creates the header for the script file.

The default implementation does nothing.

create_rules (*maxrules : int*) → str
Creates a rule list from the currently active RuleSet. Gets *maxrules* rules from the set RuleSet and passes their code back as string for the script file.

Adapted from Pieter Spronck's algorithm as explained in Spronck et al: 2005, 'Adaptive Game AI with Dynamic Scripting'.

create_script (*scriptfile : object*, *maxrules : int*)
Creates a script from the available RuleSet using the passed script file. A maximum of *maxrules* rules will be written. *scriptfile* can be either a file object or filename. In case of a file object it is assumed to be writeable and won't be closed on leaving the function (but flushed).

CHAPTER

FOUR

C++ IMPLEMENTATION

The C++ implementation of dynrules features a set of classes similar to the Python implementation and can be found in the `cplusplus/` subdirectory of the distribution.

4.1 Installation

The C++ implementation ships with an own set of build instructions that are completely independent from Python. To build (and install) it, you can use the `make` tool on Unix-like platforms

```
$ make && make install
```

and the Visual Studio.NET solution file under `win32/` on Windows platforms.

4.2 Usage

For conrete details about the API, please take a look at either the header file comments or the API documentation in `cplusplus/doc/html`.

Further readings:

CHAPTER

FIVE

RELEASE NEWS

This describes the latest changes between the dynrules releases.

5.1 0.1.0

Released on 2013-05-22

Python framework:

- Removed Python C module.
- Renamed RuleManagement class to RuleManager.
- Renamed MMapRuleManagement class to MMapRuleManager.

C++ framework:

- Renamed RuleManagement class to RuleManager.
- Renamed MMapRuleManagement class to MMapRuleManager.

5.2 0.0.15

Released on 2012-09-30

Python framework:

- Added PyPy support.
- More compatibility fixes for Python 3.x.
- Fixed a refcount issue in CRuleSet.update_weights(), which might lead to crashes.
- Moved unit tests into dynrules package.

5.3 0.0.14

Released on 2011-05-14

Python framework:

- Compatibility fixes for Python 3.2.

5.4 0.0.13

Released on 2010-05-19

Python framework:

- Fixed constructor for RuleManagement class.

5.5 0.0.12

Released on 2010-03-26

Python framework:

- Fixed file recognition issue for Python 3.x in CLearnSystem.

5.6 0.0.11

Released on 2009-12-25

- Fixed documentation distribution.

5.7 0.0.10

Released on 2009-12-24

Python framework:

- Changed CRule.id to be an arbitrary object.
- Fixed minweight/maxweight range checks for CRuleSet.
- Fixed minweight/maxweight checks within the C API interfaces.

5.8 0.0.9

Released on 2009-12-22

C++ framework:

- Changed RuleManagement.save_rules_hint_file() to take a filename instead of prefix and suffix.
- Better Makefile support
- Fixed several pointer/const/reference issues.
- Changed char* exceptions to be invalid_argument types.
- Added HTML API reference.
- Added Win32 VC++ (VS.NET 2008) support.

Python framework:

- Changed RuleManagement.save_rules_hint_file() to take a filename instead of prefix and suffix.

- Added C API documentation.
- Added C API tests.
- import cleanups.

5.9 0.0.8

Released on 2009-07-03

Python framework:

- Fixed an import bug in LearnSystem.

5.10 0.0.7

Released on 2009-03-09

C++ framework:

- Added missing RuleManagement::getMaxRules() method.
- Fixed documentation comments.
- RuleSet::updateWeights() method receives only a fitness argument now.

Python framework:

- Many fixes for correct Python 3.x support.
- CLearnSystem.create_script() now can handle file names properly.
- Added API reference to documentations.

5.11 0.0.6

Released on 2008-12-16

Python framework:

- Added Python 3.x support. Note that CLearnSystem.create_script() does not accept file names with Python 3.0, only file objects.

5.12 0.0.5

Released on 2008-11-21

C++ framework:

- Fixed a bug in RuleSet::updateWeights() which caused wrong weight results.
- Added missing MMapRuleManagement.h include to dynrules.h

Python framework:

- Fixed name ambiguity for the Python and C implementation. The visible C types in the dynrules packages were renamed to CRule, CRuleSet and CLearnSystem, the C module types still have their original name.

- Fixed C API slots.
- Added RuleSet.find() and CRuleSet.find() methods.
- New RuleManagement class.
- New MMapRuleManagement class for in-memory rule management.

5.13 0.0.4

Released on 2008-11-20

C++ framework:

- New RuleSet.find() method.
- New abstract RuleManagement class for managing rules.
- New MMapRuleManagement class for in-memory rule management.
- Changed API to pass object pointers around instead of objects.

5.14 0.0.3

Released on 2008-11-09

C++ framework:

- Added documentation.
- Fixed a minor range issue in the RuleSet constructor that allowed minweight to be smaller than maxweight.
- Fixed an int vs. double bug in the LearnSystem constructor.

5.15 0.0.2

Released on 2008-11-08

- New pure C++ framework under cplusplus

5.16 0.0.1

Released on 2008-10-06

- Initial release.

CHAPTER

SIX

TODO LIST FOR DYNRULES

- complete unittests
- add more examples

CHAPTER

SEVEN

LICENSE

CHAPTER

EIGHT

INDICES AND TABLES

- *genindex*
- *search*

CHAPTER

NINE

DOCUMENTATION TODOS

Last generated on: May 22, 2013