Reinforcement Learning for Markov Decision Processes

This is the Documentation for the code by Vlad and Elisabeth for the Practical Course "Recent Advances in Model Checking" under the supervision of Stefanie Mohr.

Some definitions:
Episode: one run through the model until the objective of the model is fulfilled (e.g. the goal is reached)
Step: one execution of an action within the model

Overview

Introduction

This project examines how well reinforcement learning for decision trees can be applied to model checking. The goal is to learn a near-optimal policy in the form of a decision tree for a given Markov Decision Process. To investigate this we reimplemented two approaches that have been published in the field of traditional Machine Learning:

1. "Decision Tree Function Approximation in Reinforcement Learning" by Larry D. Pyeatt and Adele E. Howe, published in 1998 - *The old algorithm*
2. "Conservative Q-Improvement: Reinforcement Learning for an Interpretable Decision-Tree Policy" by Aaron M. Roth, Nicholay Topin, Pooyan Jamshidi, and Manuela Veloso, published in 2019 - *The CQI algorithm*

We applied the algorithms to two different JANI models. Our results show that both algorithms almost always learn a near-optimal decision tree policy under well-chosen parameters and thus these algorithms can be applied to model checking. We did not observe a big difference in the performance of these two approaches.

Functionality

Given a Markov Decision Process modeled with the JANI format, our tool can perform the two algorithms and output a decision tree policy. For a detailed description of how to run it please refer to section Getting Started.
This Project also contains a Jupyter Notebook with a description of our tests to compare the two, the scripts to replicate them as well as plots that show the results. For more details please refer to section Appendix: Results.

Getting Started

Prerequisites

Momba

Momba describes itself as "a Python framework for dealing with quantitative models centered around the JANI-model interchange format". We use it to perform steps within a JANI model and extract information about the current state as well as possible actions. Momba can be conveniently installed by running:

```
$ pip install momba
```

For further details please refer to the official momba setup guide or user guide.

Other Requirements

The Code is written in Python 3 and has been tested with Python 3.9.0 and 3.10.0.
Apart from momba, we are also using the following dependencies:

- random
- numpy
- graphviz
- pathlib
- scipy

All of these can be installed using pip:

```
$ pip install random numpy graphviz pathlib scipy
```

We also use the built-in modules: json, operator, math, warnings and abc.

Testing your Setup

Navigate into the `src/Implementation` folder and run `algorithms.py`.
You should see something like this continuously running in your terminal:

```
Episode 1
Episode 2
...
```

This means the default algorithm run has successfully started. This will take some time as the algorithm will likely need more than 2000 Episodes to finish.
For the interpretation of the output please refer to section Interpretation of Output.

Recreating our Experiments

This code was tested on two distinct problems which are modeled as Markov decision processes:

1. The Resource Gathering problem
2. The Frozen Lake Problem

To fully understand the purpose and performance of the algorithms it is important to understand the given problems and their objectives. If you are not familiar with these problems please have a look at Appendix: Our Models.

Running the Algorithm

To run the algorithms you need to call the functions CQI or Old_Algorithm which are inside the file `src/Implementation/algorithms.py`.
Simply navigate to the `Implementation` folder and run `algorithms.py` with Python 3. The default function call that is executed by this is:

```
CQI("../Testing/models/resources_parsed_fully.jani")
```

This calls the Conservative Q Improvement algorithm from the new paper. To change to the old algorithm simply *comment out* the `CQI` function call and *comment in* the `Old_Alg` function call:

```
# CQI("../Testing/models/resources_parsed_fully.jani")
Old_Alg("../Testing/models/resources_parsed_fully.jani")
```

Both of these will run the respective algorithm on the Resource Gathering Problem.

Switching Model

To switch to the Frozen Lake model you need to change 3 things:

1. Replace `resources_parsed_fully.jani` with `lake.jani`

```
CQI("../Testing/models/lake.jani")
Old_Alg("../Testing/models/lake.jani")
```

2. Change the source file for the rewards import from `recource_rewards` to `lake_rewards` (Marked with *Important!* in the Code)

```
# from resources_rewards import get_immediate_reward, episode_finished
from lake_rewards import get_immediate_reward, episode_finished
```

3. Comment out the removal of the first variable (see below) as this is specific to only the Resource Gathering model (Marked with *Important!* in the Code)

```
# lows.pop(0)
# highs.pop(0)
# var_labels.pop(0)
```

Keep in mind that the algorithms need different parameters for each model(variation). The optimal parameters for Resouce Gathering and Lake can be found in Appendix: Optimal Parameters

Interpretation of Output

The output for the two algorithms has the same format. First, you see the algorithm counting up how many runs through the model are performed:

```
Episode 1
Episode 2
...
```

This continues, until the training and thus running of the algorithm is done. After that you should see the final output which is divided into 3 parts:

1. The learned policy decision tree
2. Information about the training that helps determine how well the algorithm performed
    1. Number of steps per episode
    2. Average reward per episode
    3. Number of leaves on the decision tree for all episodes
3. Information on how big the learned decision tree is compared to the complete decision tree

The learned decision tree is in the graphviz format and can be visualized using an online tool such as this one.

The information about training consists of three arrays. The first one is the number of steps (iterations) that it takes to reach the goal. This number should start relatively high and go down gradually. It will however always fluctuate as there is randomness during training. Please refer to the papers for more detailed information about this.
The second array shows the average reward per step during a given episode. This number should be low in the beginning and grow over time. As with the number of steps, this will always fluctuate.
The last array shows the number of leaves the decision tree has after each episode. This number is non-decreasing which means after every episode the number of leaves is greater or equal to the number of leaves before the episode. During one episode more than one leaf may be created. In the beginning, the increase should be faster than in the end.

If the numbers do not follow the above-described trends, this is an indicator that the algorithm is not working properly. You want to change the parameters of the algorithms

At last, the number of possible states for the given model is calculated. This number is the number of leaves that a complete decision tree, which represents each possible state individually, would have. The number of leaves that the learned decision tree has should be considerably lower than the number of leaves in the complete tree.

Developer’s Guide

If you want to adapt, reuse or extend this code, please refer to the following Developer's Guide. We provide a visual overview of the class structure as well as more detailed information about the most important functions and classes. The comments within the code might also help to understand how specific parts of the algorithm work.

Classes and Files

This is an overview of the class structure: TODO UML class diagram

Using New Models

For any new model you want to use, you need to implement the functions `get_immediate_reward` and `episode_finished`. We recommend creating a new file `model_name_rewards.py` which contains these functions. We provide these for the Resource Gathering problem as well as the Frozen Lake problem.
To use the algorithms on new models you have to mind a few general restrictions:

1. The model has to have the possibility to assign rewards. It does not need defined rewards, in fact, we will not use existing rewards at all. Instead, you will have to define your own rewards which may either be the ones that a model already has or ones that you come up with yourself. Models which have accepting states can use these as states with rewards as the lake problem proves.
2. The model has to have a finite number of states.
3. The model has to have a finite number of actions. All actions need to be possible to perform at all times.

Apart from this, the model has to be in the JANI format. You can find a list of tools that support conversion to JANI in the JANI Specification under Tool Support. If you are using a prism model as the source we recommend using storm within the docker container. A guide on how to set it up can be found on the official storm documentation page. You can then use storm-conv to convert the prism model to a JANI model:

```
$ /storm-conv --prism /path/tp/prims/model.pm --tojani
destination/path.jani --globalvars
```

Unfortunately, momba can not work with all JANI syntax. If you see errors when running the algorithms with your current model please have a look at Appendix: Adapting Models to fit Momba Requirements.
You also need to find good parameters for the algorithm to work properly. We recommend doing a manual grid search. We provide good parameters for our models in Appendix: Our Parameters

Important Functions

CQI()

This function implements the Conservative Q-Improvement algorithm as proposed by Roth et.al. For information about how the algorithm works please refer to the paper.

The Conservative Q Improvement function has one compulsory and 8 optional parameters. Except for the compulsory parameter `model_path`, they are parameters that influence how the decision tree is learned. They have to be changed for every single model for the algorithm to work properly. The parameters are the following:

Parameter Name	Description
`model_path`	the path to the model the algorithm is performed on

Parameter Name	Description
`epsilon`	determines the amount of randomness of choosing a random action or the action with the highest Q value. The default value is 0.5
`H_s`	starting threshold for what potential delta_Q is required to trigger a split. The default value is 8
`D`	decay for H_s. The default value is 0.9999
`gamma`	constant for the Bellman equation. The default value is 0.8
`alpha`	constant for the Bellman equation. The default value is 0.1
`d`	visit decay for Algorithm 4 and Algorithm 5(as described in the paper). The default value is 0.999
`num_of_episodes`	limit of episodes to perform during training. The default value is 10,000
`num_of_steps`	limit of steps to perform during training. The default value is 1,000,000

To learn more about the effects of these parameters, please refer to Appendix: Parameter Effects

Old_Alg()

This function implements the algorithm proposed by Pyeatt and Howe. Since the paper does not elaborate on a lot of implementation relevant details this algorithm uses a similar structure as the CQI algorithm with a different approach on when and where to split leaf nodes of the decision tree.

The function has one compulsory and 7 optional parameters. Except for the compulsory parameter `model_path`, all parameters are relevant for the performance of the algorithm and have to be adjusted to every model(variation) individually.
Old_Alg shares 6 of its seven parameters with CQI For their meanings please refer to the CQI documentation. These parameters are: `model_path`, `epsilon`, `gamma`, `alpha` (has default value 0.3 for Old_Alg), `d`, `num_of_episodes` and `num_of_steps`.

Unique to the old algorithm is `hist_min_size`, which is the minimum number of visits before a leaf node can be split. The default value is 3000.

get_immediate_rewards()

This function has to be implemented for every new model. It is provided for the Resource Gathering and Lake problems in their respective `rewards.py` files. The function takes the input parameters`old_state` and `state` which characterize the last/current state of the model and are of the momba `State` format. (TODO: more specific)
The function should return an integer value (positive or negative) depending on the state.

episode_finished()

This function has to be implemented for every new model. It is provided for the Resource Gathering and Lake problems in their respective `rewards.py` files. The function takes the input parameter `state` which characterizes the current state of the model and is of the momba `State` format.

This function should return a boolean value depending on the state. `True` indicates that the objective of the model is fulfilled and thus the episode is finished.

Important Classes

DecisionTree

A `DecisionTree` has 7 attributes:

Attribute	Function
`initial_state`	state of the model at the creation of the tree
`lows`	an array of minima of all state variables
`highs`	an array of maxima of all state variables
`action_names`	an array of names of all possible actions
`var_labels`	an array of names of all state variables
`splits`	an array of all possible splits
`actions_qs`	an array of the Q values for each possible action

It is the superclass to `DecisionTreeOld` and `DecisionTreeNew`. Functions that both these trees require such as `select_action` or `generate_splits` are implemented here.

TreeNode, LeafNode, InnerNode

`TreeNode` is an abstract class to `LeafNode` and `InnerNode`. Similar to DecisionTree, LeafNode and InnerNode are general classes that combine functionality that both algorithms require of the inner and leaf nodes. The relevant functions for this difference are `best_split` and `update`. They implement the key difference in how the decision about when and where to split. Adjustments in other functions are needed to accommodate these differences.

DecisionTreeOld, LeafNodeOld, InnerNodeOld

These classes inherit from their respective general classes. The key difference is that the leaves of type LeafNodeOld in DecisionTreeOld store a visit history called `visits`. This history is required to determine if a split is beneficial.

DecisionTreeNew, LeafNodeNew, InnerNodeNew

These classes inherit from the respective general classes. They do not have any additional attributes.

Future Work

Parameter Grid-Search

The most important thing to add in the future is an automated grid search for parameters. Currently, the user must determine the optimal parameters by manually searching for them. This can be tedious, and time-consuming and is an important source of potential human error. Using an automated grid search would eliminate the source of human error, thus ensuring that both algorithms may be compared safely as their optimal parameters have been objectively determined.

Unfortunately, an extensive grid-search, as was performed for the CQI paper for the 8 adjustable parameters of the new plus the 7 of the old algorithm was simply not feasible with the technical limitations for this practical course.

Implementation in Model Checker(s)

To use these algorithms for future work it would be beneficial to reimplement them as extensions to existing model checkers. Such integration would make the workflow much easier. It would eliminate the need to rely on momba, which is currently the biggest inhibitor to full automation of the process. Momba is not able to parse all JANI models so the user has to run our script and then still adapt the model by hand in some cases. It also doesn't allow access to rewards within the JANI model, which is why the user has to specify them manually. Both these steps would be eliminated if we were no longer reliant on momba to access a model's state information by working from within a model checker.

Appendix

Here you can find some more information about the project that is not crucial to the documentation.

Results

We compared the two algorithms in terms of immediate reward and tree size during and after training. The resulting graphs and their origins are documented inside the jupyter notebook `Simulations_Model_Checking.ipynb` located in the folder `src/Testing/Simulations`.

To display and run the jupyter notebook you need to install jupyter. You can do so using pip:

```
$ pip install notebook
```

For alternative ways to install jupyter notebook please refer to the official documentation.

Next, navigate to `src/Testing/Simulations` and run

```
$ jupyter notebook Simulations_Model_Checking.ipynb
```

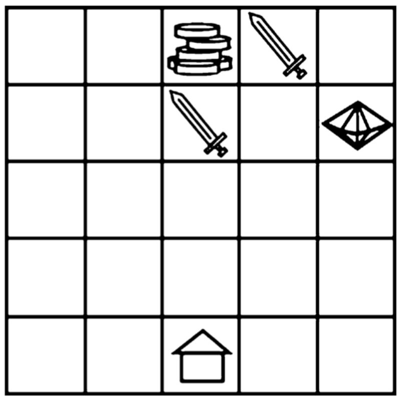
This should open a new tab in your default browser and display the notebook. To run the code within the notebook and plot the graphs you need to install pandas and matplotlib. You can do so using pip:

```
$ pip install pandas matplotlib
```

Our Models

Resource Gathering

We are given a grid environment with a home tile, a gem tile, a gold tile, and any number of enemy tiles. We use the same layout as the original problem that was proposed in Empirical evaluation methods for multiobjective reinforcement learning algorithms:



The coordinates start at (1,1) in the bottom left corner and go up to (5,5) in the top left corner. The home tile is located at (3,1), gold at (3,5), gem at (5,4) and we are facing two enemies at (3,4) and (4,5).

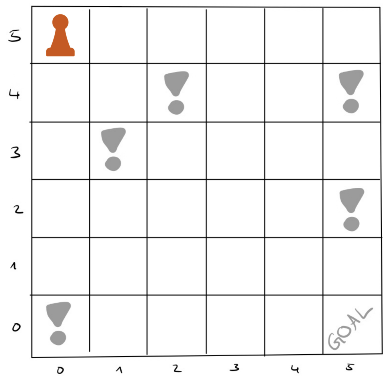
We start at our home and it is our objective to go to the gold/gem tile to pick up gold/gem and collect it by bringing it back home. It is not possible to carry more than one piece of each at the same time, however, it is possible to carry one gold and one gem simultaneously. At any given moment we have four possible actions: left, right, up, and down. If executing an action would cause us to leave the grid, the execution of this action has no effect. This means we can choose any action at all times. There are enemies which attack the player with some fixed probability if the player is on the enemy tile or one of the four tiles directly adjacent to the enemy tile.

The amount of gems and gold to collect is specified in the JANI model and can be adapted to make the problem harder or simpler (see Appendix: Adapting Existing Models). We are using a relatively complex setup with 5 gold and 3 gems to collect.

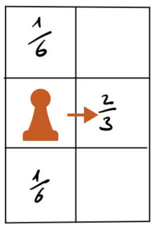
Frozen Lake Problem

We are given a grid environment with a start and an end tile. The goal is to navigate from start to finish in as few steps as possible. On arbitrary other tiles, there are holes in the ice that should be avoided.
The actions are to go east, west, north, or south. We can never move outside the given grid though.

We are using the following layout with the coordinates as indicated in the picture. The start is at (0,5) and the goal is to reach (5,0) without stepping on one of the tiles marked dangerous.



What makes this problem more complicated than others is that there is only one reward when the goal is reached and that there is a high level of randomness.
When choosing an action there is only a 2/3 chance to go where intended. There is a 1/6 chance to end up on the tiles at 90° to the sides of the intended action. The illustration below shows this phenomenon:



Adapting the Existing Models

Both the Resource Gathering and the Frozen Lake problem can be adapted to be more complicated or simpler. You can choose to either change the existing JANI model or adapt the prism model and convert it to a new JANI model. For simple alterations, we suggest looking into the JANI model while more complicated models are easier to write using the modeling language of your choice and then convert it to the JANI format.

Adapting Resource Gathering

The main modification here is to adapt the number of gold and gem to collect. As this is a fairly easy modification, it can be done within the existing JANI model.
Search for:

```
"functions": [],
"jani-version": 1,
"name": "resource-gathering",
"properties": [ ... ]
...
"type": "mdp",
"variables": [ ...
{
        "initial-value": 5,
        "name": "required_gold",
        "type": {
            "base": "int",
            "kind": "bounded",
            "lower-bound": 0,
            "upper-bound": 5
        }
    },
    {
        "initial-value": 3,
        "name": "required_gem",
        "type": {
            "base": "int",
            "kind": "bounded",
            "lower-bound": 0,
            "upper-bound": 3
        }
    }
```

Change the `initial-value` and `upper-bound` of the two constants to adapt the model. The initial value will determine after how many collected pieces the game is stopped while the upper bound is important for the algorithm to determine possible splits in the decision tree, so make sure both are the same.
Changing the position of the tiles or even the size of the field is more complicated and should not be done manually within the JANI file. The provided prism source model however can be adapted more easily by changing the respective formulas. After adopting the prism model you have to convert it which can be done with the storm converter using the command

```
./storm-conv --prism path/to/recourse-gathering.v2.pm --tojani
/destination/path/resource_gathering_adapted.jani --globalvars
```

You will now need to adapt your model to fit mombas requirements. More details on how to use storm for this purpose can be found at Using New Models.

Adapting Frozen Lake

The only information that we use from the JANI model is the location on the grid (ie field size, start position, and current position). The information about the dangerous tiles as well as the goal is only accessed through the rewards. As momba doesn't allow access to rewards, we defined those manually in `lake_rewards.py`. You can change them there as described in section get_immediate_rewards.

To change the starting position, the size of the field, or the level of randomness when taking a step, you can adapt the provided `lake.prism` file and convert it again, as is described in Adapting Resource Gathering.\
The converted Frozen Lake model does not need any adaptions to fit momba.

Adapt JANI Models to fit Momba Requirements

Momba is unfortunately not able to work with all JANI models. We found two problems with correct JANI syntax, that momba is not able to parse:

1. Function calls
2. References to constants for Variable bounds

Replacing Function Calls

This is the far bigger task of the two and we wrote a script to automate it. To use it on your own model navigate to `src/Implementation/jani_parsing.py` and change src_filepath and dst_filepath to match your model. After that simply run:

```
$ python3 jani_parsing.py
```

This will replace all function calls within the JANI model with the explicit function definitions.

Replacing Constants

Unfortunately, you still have to replace references to constants that are for example used for variable bounds. As these constants could be used in a variety of contexts and it is unclear in which contexts momba struggles to parse them, it would be hard to guarantee correctness. Therefore this part remains manual.
Simply search for any references to constants and replace them with the actual number that they represent. For Resource Gathering we had to replace `GOLD_TO_COLLECT` and `GEM_TO_COLLECT` twice each.

Our Parameters

We tested many combinations of parameters, but the following ones worked better than the others. The models here are the models described in Our Models. RG refers to the Resource Gathering Model and FL to the Frozen Lake problem.

Parameter	CQI for RG	Old Alg for RG	CQI for FL	Old Alg. for FL
`epsilon`	0.5	0.5	0.5	0.5
`H_s`	8	-	8	-
`D`	0.9999	-	0.9999	-
`gamma`	0.8	0.8	0.8	0.8
`alpha`	0.1	0.3	0.1	0.3
`d`	0.999	0.999	0.999	0.999
`num_of_episodes`	10000	10000	10000	10000
`num_of_steps`	1000000	1000000	100000	100000
`hist_min_size`	-	3000	-	3000

Parameter Effects

The following table gives the reader the idea of how changing each individual parameter influences the algorithm's performance.

Parameter	Effect
`epsilon`	Higher epsilon increases the probability of taking a random step, has to be between 0 and 1
`H_s`	The higher, the better the overall performance given enough episodes
`D`	The closer to 1, the better the overall performance, given enough episodes
`gamma`	The closer to 1, the more the exponential moving avg. leans towards new values. Impact varies depending on other parameters
`alpha`	The closer to 1, the more the exponential moving avg. leans towards new values. Impact varies depending on other parameters
`d`	The closer to 1, the better the overall performance given enough episodes
`num_of_episodes`	The higher, the longer the model trains, hence, better avg. reward per episode, but higher amount of tree nodes
`num_of_steps`	The higher, the longer the model trains, hence, better avg. reward per episode, but higher amount of tree nodes
`hist_min_size`	The higher, the better the overall performance, given enough episodes