



ulm university universität
uulm

**Fakultät für
Mathematik und
Wirtschafts-
wissenschaften**
Institut für numerische
Mathematik

C++ Library for Solving Markov Decision Processes

Prüfungsleistung für die Universität Ulm

Vorgelegt von:

Adrian Lauber
adrian.lauber@uni-ulm.de
1042606

Gutachter:

Prof. Dr. Andreas Borchert

2020

Fassung August 11, 2020

© 2020 Adrian Lauber

Satz: PDF- \LaTeX 2_ε

Contents

1	Introduction	1
2	Markov Decision Process	2
2.1	Definition	2
2.2	Algorithms	2
3	Implementation	4
3.1	Design goals	4
3.2	Interface	5
3.3	Model	6
3.4	Modules	6
3.5	Module factory	7
3.5.1	Constructor	8
3.5.2	Factory	8
3.6	Array3D	8
3.7	Build	8
3.8	Testing	9
4	Extensibility	10
4.1	Modules	10
4.2	Partial Observability	10
4.3	Reinforcement Learning	10
	Bibliography	12

1 Introduction

The goal of this project is to implement a library to solve *Markov decision processes (MDPs)*. The focus is on extensibility, facilitating modern C++ features like smart pointers as well as following well-established C++ guidelines and best-practices.

The first chapter is a brief introduction into the problem class for which the library functionality is developed.

In the second chapter the focus is on the actual implementation. The class hierarchy is described as aspects of the build and test environment.

2 Markov Decision Process

Markov Decision Processes describe a time-discrete and stochastic process that can be used to model different decision-making problems. MDPs are used in all kinds of domains and multiple algorithms have been proposed to solve a particular MDP.

2.1 Definition

A MDP is defined by a state-transition-function $P_a(s, s')$ that gives the probability of transferring to a consecutive state s' when in state s and applying action a . For a given state s and action a the probability is independent from former states or actions. This property is referred to as the *Markov property*. The goal of optimization is a scalar value referred to as the *cost* or *reward* which is either minimized (in case a problem defined by costs) or maximized (for rewards). The reward/cost is a function of the current state s and consecutive state s' given an action: $R_a(s, s')$

Solving a MDP requires to find a mapping from each state to an action that maximizes reward or minimizes the costs. This mapping is referred to as the *policy* and is denoted by π .

2.2 Algorithms

For finite state and action spaces algorithms based on dynamic programming are a popular choice for solving MDPs. The foundation for applying dynamic programming to this problem class is the definition of a recursive function that assigns a value to a state given that a specific policy π is applied.

$$V_{\pi} = \sum_a \pi(a | s) \sum_{s', r} p(s', r | s, a) [r + \gamma v_{\pi}(s')] \quad (2.1)$$

Equation 2.1 is referred to as the *Bellmann equation for value functions*. γ is the *discount rate* parameter: $0 \leq \gamma \leq 1$

A common algorithm is referred to as *policy iteration*. In an iterative cycle the value function of each state following a policy is first evaluated (*policy evaluation*) and then the policy updated (*policy improvement*). This process is depicted in Figure 2.1.

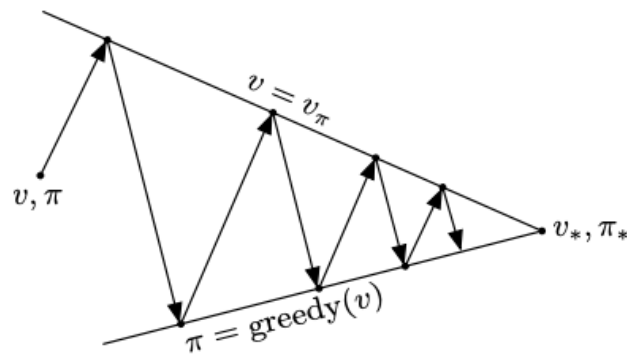


Figure 2.1: Iteration between policy evaluation and policy improvement [3]

3 Implementation

3.1 Design goals

Considering potential use cases is an important step when creating the interface and architecture of a library. The purpose of the developed library is to solve a defined decision making problem and in order to do that following input is needed:

- Definition of the Markov decision process
- Configuration and parameters for solving

For the definition of the MDP there are two main use cases. It has either been formally described in a file or it lives within an application. For the first use case it has been decided that the library shall provide functionality to parse a MDP from a file and that additional parsers for other formats can be easily added. For the other use case the library shall expose the model class in the interface so that an external application can create and setup a MDP and then pass it to the library for solving. For this use case a wrapper class would have been an alternative to directly exposing the model class. This alternative has drawbacks due to the different representations possible and also the sizes of the data structures that need to be defined. For example providing a wrapper method to set the probability value of a single state transition for a given action would require infeasible many calls for a MDP with a large state and action space.

A structure containing all necessary parameters shall be exposed in the interface so that the application can setup the structure to its needs and then pass it for solving.

After solving the typical use case is to analyze the resulting strategy and/or deploy it to the environment it has been created for. This requirement shall be fulfilled by transferring the ownership to an object containing the policy. Owning this object

shall enable the application to draw actions for a given state. This requirement is especially crucial for stochastic policies, where the policy object also depends on random number generators. Providing the application with the ownership of a policy enables easy deployment of the optimized strategy. An alternative would have been the transfer of a data structure defining the resulting stochastic or deterministic policy. In case of a stochastic policy this can be a heavily complex data structure (for example when using neural networks as a policy) and has consequently not been considered.

Since many different solving methods exist, new functionality shall be easy to integrate.

3.2 Interface

Considering the requirements from section 3.1 the header *interface.hpp* is provided to the application:

```
struct Params{
    std::string mdp_filepath_;
    std::string module_parser_;
    std::string module_eval_;
    std::string module_policy_;
    std::string module_solver_;
    std::size_t solver_iteration_cnt_;
};
/*
Solve mdp that is defined in a file on the filesystem
*/
std::unique_ptr<Policy> solve_filebased_mdp(const Params& params);
/*
Solve mdp that has been defined by the application
*/
std::unique_ptr<Policy> solve_external_mdp(Model& model,const Params&
    params);
```

3.3 Model

The model class is the core element of library because it holds information about the MDP to be solved. It is initialized using the default constructor and a Parser-Module (section 3.4) is parsing the MDP definition from the input file to the model class. Since memory for the model class members has to be allocated dynamically, the parsing model class object offers a public method to allocate the required memory for the object. Once the Parser-Module has determined the size of the state and action space, *setArrays()* can be called to initialize the array data structures of the state transition matrix and the reward matrix. A method to validate the model data is offered by the model class and is called by the Parser-Module to abort parsing if an inconsistency has been detected.

The state transition matrix and reward matrix are represented using the *Array3d template class* (section 3.6). Both matrices are used to look up transition probabilities and rewards.

$$P_a(s, s') = state_transition_matrix[a, s, s'] \quad (3.1)$$

$$R_a(s, s') = reward_matrix[a, s, s'] \quad (3.2)$$

3.4 Modules

In order to achieve a high level of extensibility this base class acts a blue print for classes to read or write the model. Using this common interface enables easy integration which is described in detail in section 3.5. The module class consists of a protected constructor to set its most important member: a reference to the model.

Four interfaces inherit from the module class and describe different types of modules:

- Parser
- Policy

- Evaluation
- Solver

The abstract *Parser* class acts as an interface for concrete Parser implementations. There exist several file formats to describe MDPs. For the sake of a running example, a rudimentary implementation to parse *POMDP* file format [2] is provided.

The abstract *Policy* class as a interface for different variations of *policy improvement*. The *policy mapping* member is a multidimensional matrix represented as an *Array3d Object* (section 3.6) in order to both support deterministic policies and stochastic policies. A method is provided to select an action given a certain state. This is needed by Evaluation-implementations. The policy class is also used as a return type for the external interface of the library. This way an external application can use the reference to directly integrate the optimal strategy for the environment.

The abstract *Evaluation* class is an interface for different implementatons of policy evaluation. A simple form of policy evaluation is provided in the library: *iterative policy evaluation*. This method applies Equation 2.1 to update the value of each state.

The *Solver* interface and base class provides a general solve method. Solving algorithms combine policy evaluation and policy improvement to find the optimal policy. *Policy iteration* is implemented as an example.

3.5 Module factory

In order to minimize the amount of effort to integrate for example an additional Parser or Solver, a factory is implemented. This enables integration without adding any references in existing code. To ensure scalability for

For each new module two classes are necessary.

- Class with new functionality, inheriting from Module Class
- Constructor class, inheriting from Constructor template

asd

```
template <typename T>  
concept ModularType = std::is_base_of<Module, T>::value;
```

3.5.1 Constructor

Each new module

3.5.2 Factory

3.6 Array3D

To process the state transition matrix and reward structure of a MDP, a multidimensional array implementation is needed. One option is to use `std::vector` holding another `std::vector` for the two-dimensional case. The `std::vector` class manages its own resources following RAII (Resource Acquisition Is Initialization) making this a simple and robust solution but performance-wise there is a drawback. The memory for this nested structure will be fragmented which will slow down access. In order to make use of `std::vector` without causing fragmented memory the `Array3d` template class acts as a wrapper to a one-dimensional `std::vector`. The `()` operator accepts three indices and maps those to the 1d `std::vector` member. The wrapper performs a boundary check on the 3d indices. If successful, the access to the 1d vector can be performed without the need for a boundary check. This further enhances the performance.

3.7 Build

A makefile is used for compiling the library and linking it to the example (binary: example) as well as the unit tests (binary: tests). The compiler is set to `-std=c++20` since *Concepts* are a *C++20* feature.

Due to the modular concept, the concrete implementations are not referenced anywhere in the rest of the codebase. Since registration of all modules is done via instantiating unreferenced, static variables, the linker has to be told to not drop any unreferenced symbols. This is achieved using the *-whole-archive* flag. The linker on macOS requires a different flag. This is considered by setting an OS variable when executing make: *make OS=osx*.

3.8 Testing

Catch2[1] has been chosen as a test framework. It is a header only library which eases integration. Test cases for each class are located in the same directory with a *_t.cc* suffix. The build process generates an executable containing all tests in the *bin* directory. Overall 48 assertions in 8 different test cases are evaluated. Testing the *concepts and constraints* is split up from the overall testing since the expected outcome is a compile-time error. Manual compilation and review of following file is required to validate the introduced concept: *concepts_t.cxx*.

4 Extensibility

4.1 Modules

The concept of modules enables to easily integrate additional functionality. By forcing new classes to inherit from module, information that is needed in all modules can be easily added, for example an internal version information. The integrated module categories (Solver, Policy,...) provide interfaces in order to avoid any changes in the existing code base.

4.2 Partial Observability

One variant of MDPs are with partial observable Markov decision processes, abbreviated by *POMDP*. In addition to a MDP, the state cannot be directly observed. Instead *observations* are accessible which can be assigned to a certain state with a probability. By inheriting from the Model class a POMDP class can be easily integrated.

4.3 Reinforcement Learning

Reinforcement learning problems can also be defined as a MDP but with (at least partially) unknown state transition behavior. A common approach is to first learn the state transition behavior by drawing samples from an environment and then solving the defined MDP. Reinforcement learning algorithms can be added to the Solver module category. A new module category with different environment imple-

mentations can manage sample data that is generated by rolling out a policy on the defined environment. This extension is compatible to the current architecture.

Bibliography

- [1] *Catch2 test framework*. 2020. URL: <https://github.com/catchorg/Catch2>.
- [2] *POMDP file format*. 2020. URL: <http://pomdp.org/code/pomdp-file-spec.html>.
- [3] Richard S. Sutton and Andrew Barto, eds. *Reinforcement learning: an introduction*. Adaptive computation and machine learning. Cambridge, Mass. [u.a.]: MIT Press, 2018, XVIII, 322 Seiten. ISBN: 9780262039246.