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 13, 2020

Satz: PDF-LaTeX 2$_\varepsilon$

Contents

1 Introduction

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

The second chapter is a brief introduction into MDPs for which the library functionality is developed.

The third chapter describes important use cases for the library and requirements that are derived from those.

In the fourth chapter the focus is on the actual implementation. The library interface, class hierarchies and design patterns are described that are used to meet the requirements. Also the exemplary application is introduced. Building and testing the library is part of the following chapter.

The last chapter is devoted to considerations about potential extensions of the library and how they could be integrated.

2 Markov Decision Process

Markov Decision Processes describe a time-discrete and stochastic process that can be used to model different planning and 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:

- set of states (S)
- set of actions (A)
- state-transition function $P_a(s, s')$
- reward-function $R_a(s, s')$

The state-transition-function $P_a(s, s')$ 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. A mapping is referred to as the *policy* and is denoted by π. A policy that achieves the highest possible reward is referred to as the *optimal policy*.

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.

$$V_\pi = \sum_a \pi(a \mid s) \sum_{s',r} p\left(s', r \mid s, a\right) \left[r + \gamma v_\pi\left(s'\right)\right] \tag{2.1}$$

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.

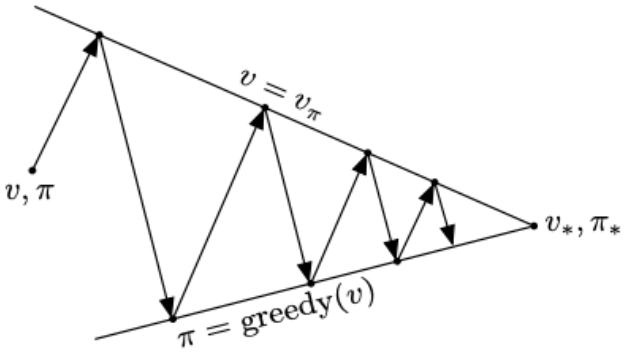


Figure 2.1: Iteration between policy evaluation and policy improvement [7]

For both policy evaluation and policy improvement different approaches exist. When using *iterative policy evaluation* and a *greedy* policy update, policy iteration is guaranteed to converge to the optimal policy. Iterative policy evaluation describes the evaluation of the current policy using Equation 2.1. A greedy policy update is selecting the action with the highest expected reward according to a one-step look-ahead using the value function of a consecutive state.

3 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.

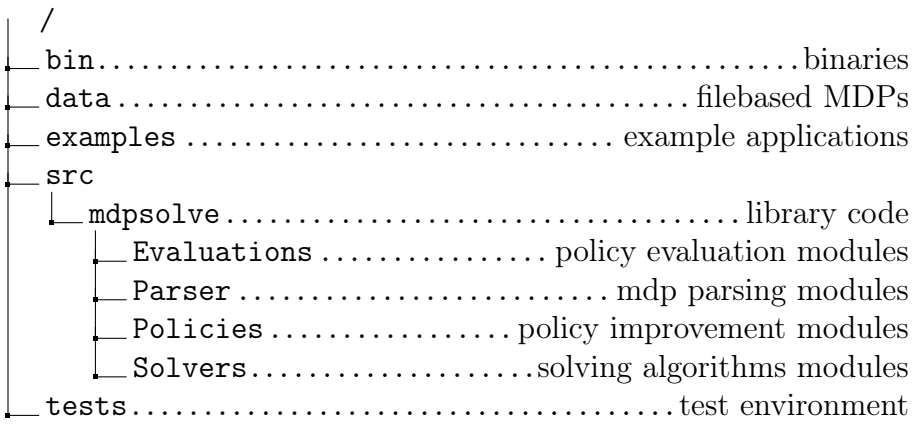
After solving the typical use case is to analyze the resulting strategy and/or deploy it to the environment it has been created for. This requirement shall be fulfilled by transferring the ownership to an object containing the policy. Owning this object shall enable the application to draw actions for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

A typical use case is to evaluate different solvers and solver configurations on the same model to compare the performance. This 1:n relationship shall be reflected in the architecture. Since many different solving methods exist, new functionality shall be easy to integrate.

The library shall guarantee *basic exception safety*. To ensure high code quality a common C++ guideline has been followed [2] as well as a style guide for naming conventions [5]. *clang-format* has been used for code formatting with the settings from the Linux kernel [8]. The project layout has been chosen according to another guideline [3].

4 Implementation



4.1 Interface

Considering the requirements from chapter 3 the header *interface.hpp* shall be 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);
```

The functionality provided in the interface is implemented in *solve.cpp*.

4.2 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 4.3) 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 4.5). Both matrices are used to look up transition probabilities and rewards.

$$P_a(s, s') = state\_transition\_matrix[a, s, s'] \tag{4.1}$$

$$R_a(s, s') = reward\_matrix[a, s, s'] \tag{4.2}$$

4.3 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 4.4. The module class consists of a protected constructor to set its most important member: a reference to the model. This class is totally independent from the new C++20 feature which is also called Modules.

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

- Parser (parser.hpp)
- Policy (policy.hpp)
- Evaluation (evaluation.hpp)
- Solver (solver.hpp)

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 [6] is provided. This format is referred to in the code by the name of the author: *Cassandra*. This class only parses a minimal subset of the POMDP grammar and lacks validation.

The abstract *Policy* class as a interface for different variations of *policy improvement*. The *policy mapping* member is a multdimensional matrix represented as an *Array3d Object* (section 4.5) 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.

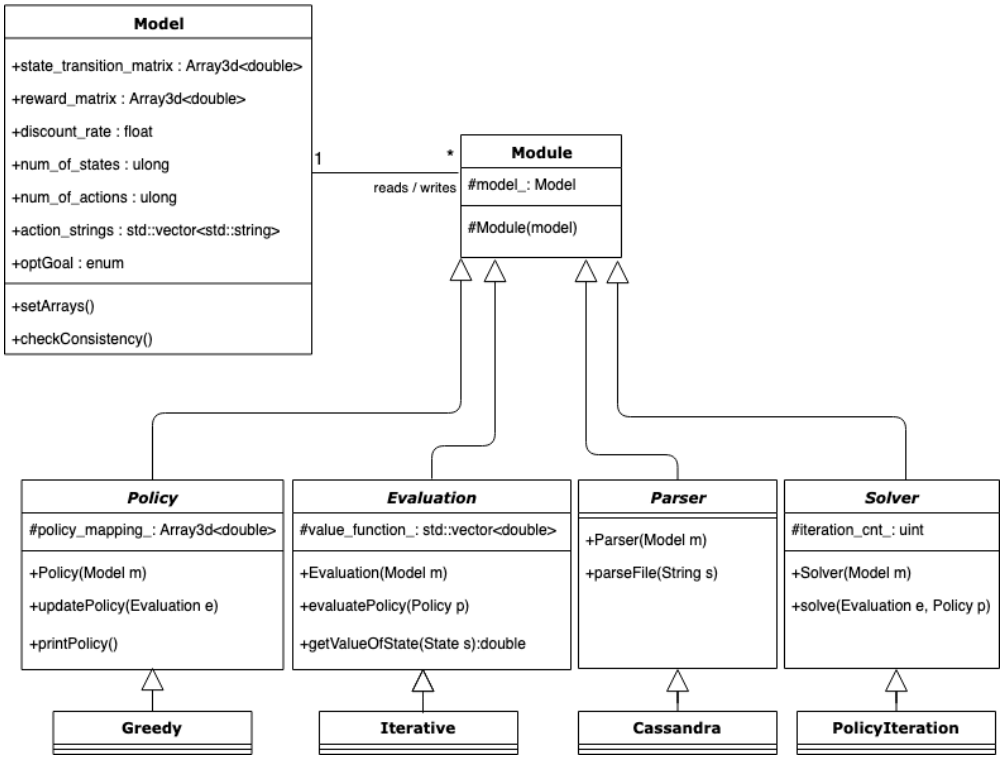


Figure 4.1: Model and Modules

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

Classes implementing the required interface of these interfaces are located in the according subdirectories.

4.4 Module factory

In order to minimize the amount of effort to integrate and instantiate additional functionality, a factory (*factory.hpp*) is implemented. The implementation for the factory and constructor is based on the lecture material [1]. A factory pattern combined with self registering types enables integration of new functionality without adding any references in existing code. To ensure scalability it has been decided to use a template class as a factory and instantiate a factory for each module category

(Parser, Policy, Evaluation, Solver). Based on the factory template, customized factories can be implemented if needed. The factory template class is designed as a singleton and follows the *construct on first use idiom*. A given name defined in the constructor of a new module is used as an identifer within the factory to decide which object to create. Using multiple factories reduces the risk of name conflicts between modules. Alternatively a global factory could have been used with the category name encorporated in the given name. A global factory would have offered lower flexibility and was disregarded.

The constructor (*constructor.hpp*)is a template class and used as a type in the factory as well as a base class for concrete constructors. The concrete constructor class has to override the virtual create(..) method and register to its factory. In order to minimize necessary code changes when including a new type, it has been decided to implement self registering types by instantiating a static constructor object. This has drawbacks when it comes to building since the static objects and new functionality remains unreferenced and dropping it has to be avoided as described in section 5.1. It has been decided that the advantage of not having to touch any existing code overweighs.

Figure 4.2 depicts a factory of type Solver and a new Solver implementation ("NewSolver").

For each new module two classes have to be added:

- Class with new functionality, inheriting from module class or a module category
- Constructor class, inheriting from Constructor template class

After the constructor is succesfully registered, the factory can call its create(..) method which returns a pointer to the new object. In order to avoid memory leaks, the factory creates a std::unique_ptr object for the returned pointer. The factory then returns this std::unique_ptr to the caller which will also transfer exclusive ownership. By using a unique_ptr the caller does not have to handle deletion of the created object. The drawback of this solution is that if the caller happens to need a std::shared_ptr for distribution the construction of it comes at the cost of a dynamic memory allocation. Since the unique_ptr is more lightweight and no use case for a shared_ptr could be identified, it has been favored.

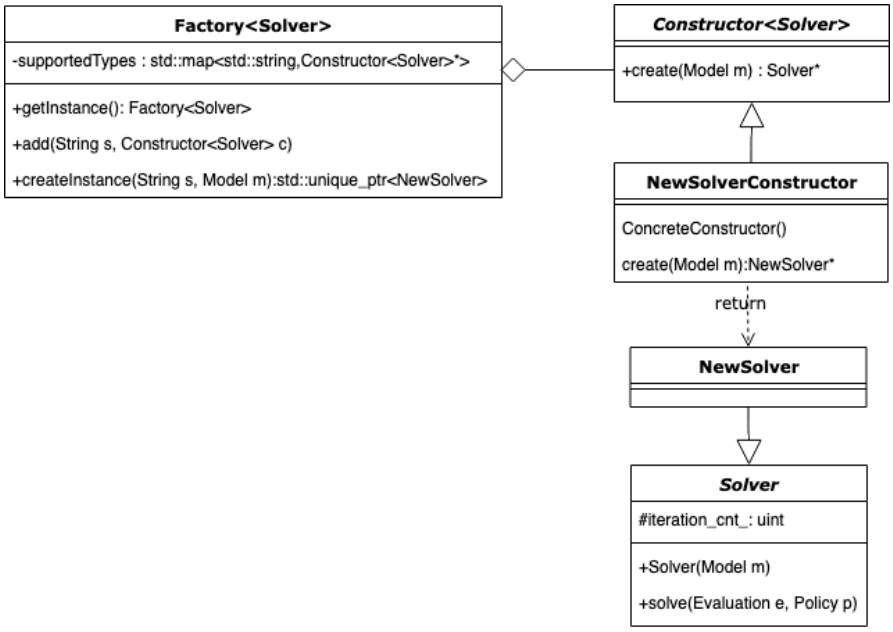


Figure 4.2: Solver factory

Both constructor and factory enforce that new classes inherit from the module class when using the module factory. This is achieved with a new C++20 feature named *concepts*. Concepts ease the implementation of compile-time validation of template arguments and provide meaningful compile errors to the user. Following concept enforces the template argument to inherit from the module class:

---

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

The concept of a *ModularType* is used for the factory and constructor template classes.

4.5 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

-0.04	-0.04	-0.04	+1
-0.04	#	-0.04	-1
-0.04	-0.04	-0.04	-0.04

Table 4.1: 4x3.95.POMDP by Stuart Russel

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 (*array3d.hpp*) 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 enhances the performance.

The Array3d template class does not constrain the template argument in any way. Since internally a std::vector object with the same type has to be instantiated, the restrictions that the vector class provides are sufficient.

4.6 Example

A sample mdp is provided in the *data* directory: "4x3.95.POMDP" [6]. The mdp describes a 4x3 gridworld (Table 4.1) with a blocked cell (#), a cell with high positive reward(+1) and a cell with high negative reward(-1). The cell field is not in the state space and the states of the cells with high rewards cause the transition to a random state (restarting). The action space consists of four move actions: north(N), south(S), east(E), west(W). Moving into the wall or into the blocked cell will return the same state. With a probability of 20% an move into a direction which is perpendicular to the intended direction is happening. The mdp is used with full observability.

The example application (directory: *examples*) performs 20 steps of policy iteration on the defined mdp and then prints the expected optimal policy (Table 4.2).

E	E	E	N
N	#	N	N
N	E	N	W

Table 4.2: Solution of the example mdp

5 Environments

5.1 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). Binaries can be found in the *bin* directory. The compiler is set to *-std=c++20* since *Concepts* are a *C++20* feature. Binaries have to be executed in the main directory (./bin/example) since relative paths are used for solving a mdp from file.

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*. Using this linker option causes the binary to be bloated. To reduce the bloating effect a separate library could be build only consisting of modules.

Manual registration would not need any of the mentioned measures and should be considered when releasing an application.

5.2 Testing

Catch2[4] 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 62 assertions in 9 different test cases are evaluated. Testing

the *concepts and constraints* is split up from the overall testing since the expected outcome is a compile-timer error. Manual compilation and review of following file is required to validate the introduced concept: concepts_t.cxx.

6 Extensibility

6.1 Modules

The concept of modules enables to easily integrate additional functionality. By forcing new classes to inherit from module, information that is common over 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.

6.2 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.

6.3 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] Prof. Andreas Borchert. “Vorlesungsbegleiter zu Objektorientierte Programmierung mit C++ Kapitel 5: Design-Patterns und Techniken zum dynamischen Polymorphismus”.

[2] *C++ Core Guidelines*. 2020. URL: https://github.com/isocpp/CppCoreGuidelines.

[3] *C++ project layout guide*. 2020. URL: https://github.com/vector-of-bool/pitchfork.

[4] *Catch2 test framework*. 2020. URL: https://github.com/catchorg/Catch2.

[5] *Google C++ Style Guide*. 2020. URL: https://google.github.io/styleguide/cppguide.html.

[6] *POMDP file format*. 2020. URL: http://pomdp.org/code/pomdp-file-spec.html.

[7] 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.

[8] *clang-format style of Linux Kernel*. 2020. URL: https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/.clang-format.